MANNING



# C++ 并发编程实战

C++ Concurrency IN ACTION

〔美〕Anthony Williams 著

周全 梁娟娟 宋真真 许敏 译

人民邮电出版社
POSTS & TELECOM PRESS

MANNING

# C++ 并发编程实战

C++ Concurrency IN ACTION

〔美〕Anthony Williams 著

周全 梁娟娟 宋真真 许敏 译

人民邮电出版社

北京

**图书在版编目（CIP）数据**

C++并发编程实战 / (美) 威廉姆斯 (Williams, A.) 著 ; 周全等译. -- 北京 : 人民邮电出版社, 2015.6(2017.4重印)
ISBN 978-7-115-38732-5

Ⅰ. ①C… Ⅱ. ①威… ②周… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第069669号

**版权声明**



♦ 著　　　　[美] Anthony Williams
　译　　　　周 全　梁娟娟　宋真真　许 敏
　责任编辑　陈冀康
　责任印制　张佳莹　焦志炜
♦ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　固安县铭成印刷有限公司印刷
♦ 开本：800×1000　1/16
　印张：31.75
　字数：686 千字　　　　2015 年 6 月第 1 版
　印数：5 501－5 800 册　　2017 年 4 月河北第 6 次印刷

著作权合同登记号　图字：01-2009-3542 号

定价：89.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

# 内容提要

本书是一本基于 C++11 新标准的并发和多线程编程深度指南。内容包括 std::thread、std::mutex、std::future 和 std::async 等基础类的使用，内存模型和原子操作、基于锁和无锁数据结构的构建，以及并行算法和线程管理，最后还介绍了多线程代码的测试。本书的附录部分还对 C++11 新语言特性中与多线程相关的项目进行了简要的介绍，并提供了 C++11 线程库的完整参考。

本书适合于需要深入了解 C++多线程开发的读者，以及使用 C++进行各类软件开发的开发人员、测试人员阅读。使用第三方线程库的读者，也可以从本书后面的章节中了解到相关的指引和技巧。同时，本书还可以作为 C++11 线程库的参考工具书。

# 序

我是在离开大学后的第一份工作中遇到多线程编程的概念的。我们当时在编写一个数据处理应用程序，它需要用传入的数据记录来填充数据库。虽然数据很多，但每个数据都是独立的，并且在它被插入数据库前需要大量的处理。为了充分利用 10-CPU UltraSPARC 的能力，我们在多线程中运行代码，让每个线程处理它们自己的一组输入数据。在这个过程中，我们用 C++编写代码，使用 POSIX 线程，并且犯了相当多的错误。尽管多线程对我们而言都是全新的，但我们最终还是完成了。正是在这个项目中的工作，我第一次了解了 C++标准委员会和全新发布的 C++标准。

我对多线程和并发有前所未有的兴趣。虽然别人认为它困难、复杂，是问题之源，但我却视它为强大的工具，因为它可以允许你利用现有的硬件让代码运行得更快。随后我了解到它即便是在单核硬件上也能提高应用程序的响应和性能，通过使用多线程来隐藏诸如 I/O 这样耗费时间的操作延迟。我还了解了它怎样在 OS 级别工作以及 Intel CPU 如何处理任务切换。

同时，我对 C++的兴趣给我带来了与 ACCU 以及 BSI 的 C++标准专家组以及 Boost 接触的机会。我凭兴趣跟进了 Boost 线程库的初期开发，当它被最初的开发者抛弃时，我便趁机介入了。我从此成为了 Boost 线程库最主要的开发者和维护者。

当 C++标准委员会的工作从修正现有标准的缺陷转移到编写下一代标准（希望在 2009 年之前完成故命名为 C++0x，现在正式为 C++11，因为最终发布于 2011 年）的提案后，我更多地参与到 BSI 并且开始起草我自己的提案。多线程刚被明确地提上议事日程，我就全力以赴地投入进去，并且撰写或共同撰写了许多与多线程和并发相关的提案，它们组成了新标准的一部分。我感到很荣幸，可以有这个机会用这种方式组合我的计算机相关的两大兴趣——C++和多线程。

这本书借鉴了我在 C++和多线程上的全部经验，其目标是教会其他 C++开发者如何安全并有效地使用 C++11 线程库。我也希望能顺带传授一些我在这方面的热情。

# 译者简介

周全，软件工程师，毕业于中国科学技术大学信息学院，现就职于中国人民银行合肥中心支行科技处。有着丰富的系统集成和运维经验，对虚拟化也有较深入的研究，曾从事.NET 开发和培训工作。读者可以通过 Email:zhouquan.pbc@foxmail.com 与他联系。

梁娟娟，2010 年毕业于中国科学技术大学信息技术学院，现就职于中国人民银行合肥中心支行。

宋真真，网络工程师，2008 年毕业于合肥工业大学计算机与信息学院，现就职于中国人民银行合肥中心支行科技处，参与软件开发、项目管理等工作，爱好数据库、编程等研究。读者可以通过 Email: hfut_szz@sina.com 与她联系。

许敏，软件工程师，2005 年获得软件测试工程师证书。现就职于中国人民银行合肥中心支行科技处，负责项目管理工作。读者可以通过 Email: xu_min@sina.com 与她联系。

致 *Kim*、*Hugh* 和 *Erin*。

# 致谢

我要首先对我的妻子 Kim 说一声“谢谢”，感谢在我写这本书的时候她给我的爱和支持。写书占用了我最近 4 年很多的空闲时间，没有她的耐心、支持和理解，我不可能做到。

其次，我想感谢 Manning 的团队，包括总编 Marjan Bace、副总编 Michael Stephens、开发编辑 Cynthia Kane、编审 Karen Tegtmeyer、文字编辑 Linda Recktenwald、校对 Katie Tennant 和制作经理 Mary Piergies。他们使得这本书成功面世，没有他们的努力，你现在就读不到这本书了。

我还想感谢 C++标准委员会的其他成员，他们为多线程工具上撰写了文件。正是 Andrei Alexandrescu、Pete Becker、Bob Blainer、Hans Boehm、Beman Dawes、Lawrence Crowl、Peter Dimov、Jeff Garland、Kevlin Henney、Howard Hinnant、Ben Hutchings、Jan Kristofferson、Doug Lea、Paul McKenney、Nick McLaren、Clark Nelson、Bill Pugh、Raul Silvera、Herb Sutter、Detlef Vollmann 和 Michael Wong，以及所有在文件上批注的人们，他们在委员会会议上进行了讨论，还有其他人的帮助才形成了 C++11 中的多线程和并发支持。

最后，我还想感谢下面的人：Dr. Jamie Allsop、Peter Dimov、Howard Hinnant、Rick Molloy、Jonathan Wakely 和 Dr. Russel Winder，他们的建议极大地改进了这本书。另外特别感谢 Russel 的详细审阅，还有技术校对 Jonathan，在编辑出版过程中极其用心地检查了终稿中所有内容中的错误（当然所有的错误都是由我造成的）。此外我想感谢我的审稿专家组：Ryan Stephens、Neil Horlock、John Taylor Jr.、Ezra Jivan、Joshua Heyer、Keith S. Kim、Michele Galli、Mike Tian-Jian Jiang、David Strong、Roger Orr、Wagner Rick、Mike Buksas 和 Bas Vodde。还要谢谢 MEAP 版的读者，他们花费时间来指出错误或是强调了需要阐明的部分。

# 前言

这本书是对新 C++标准中的并发和多线程工具的深度指南，内容包括从 std::thread、std::mutex 和 std::async 的基本用法，到复杂的原子操作与内存模型。

## 路线图

前 4 章介绍了由类库提供的各种类库工具以及他们如何使用。

第 5 章涵盖了内存模型和原子操作的低阶基础，包括原子操作怎样在其他代码上强制实行排序约束，并标志着导言章节的结束。

第 6 章和第 7 章开始涵盖高阶的论题，包括一些如何使用基础工具来构造更复杂的数据结构的示例——第 6 章中基于锁的数据结构，以及第 7 章中无锁的数据结构。

第 8 章继续高阶论题，包括如何设计多线程代码的指南，涵盖了影响性能的论点，以及各种并行算法的示范实现。

第 9 章涵盖了线程管理——线程池、工作队列和中断操作。

第 10 章包括了测试和调试——bug 的类型，定位它们的技巧，如何测试它们，等等。

附件包含了对由新标准引入的与多线程相关的一些新语言工具的简要介绍，第 4 章中提到的消息传递库的具体实现，以及 C++11 线程库的完整参考。

## 谁应该阅读本书

如果你打算用 C++编写多线程代码，你就应该阅读本书。如果你正要使用 C++标准库中新的多线程工具，这本书是必备的指南。如果你正使用替代的线程库，后面几章中的指引和技巧应该也是有用的。

假设你对 C++已经有了很好的了解，但对新的语言特性却不甚熟悉，这些在附录 A 中也能找到答案。假定你之前没有多线程编程的知识和经验，那就更应该阅读本书。

## 如何使用本书

如果你以前从未写过多线程代码，我建议你按顺序从头到尾阅读本书，可以跳过第5章中的细节部分，但第7章大量依赖第5章中的材料，所以如果你跳过了第5章，你一定要阅览第7章，除非你曾读过。

如果你之前未曾使用过C++11语言工具，在你开始确定准备快速开始书中例子之前最好浏览一下附录A。新语言工具的使用凸显在文字之中，然而，当你遇到了之前没有见过的东西时，总是可以翻看附录的。

如果你在其他环境中拥有大量编写多线程代码的经验，开始的几章可能让你值得浏览一遍，以便你可以看看你了解的工具怎样映射到新C++标准中。如果你打算用原子变量做一些低阶的工作，第5章就是必需的。为了确认你熟悉多线程C++中类似异常安全的东西，值得阅览一下第8章。如果你在脑海中有特定的任务，索引和目录可以帮助你快速找到相关的章节。

一旦你打算促进C++线程库的使用，附录D应该仍然有用，比如查询每个类和函数调用的细节。你可能会想一次又一次地翻回主章节，来刷新你对某一概念的使用或者看一看示例代码。

## 代码约定和下载

所有代码清单和正文中的源代码，出现等宽字体（`like this`），是为了便于从常规文本中区分出来。代码注解伴随着很多清单，指出重要的概念。在有些情况下，数字符号往往指向清单后面的注释。

本书中所有的工作示例源代码可以在出版社网址下载，www.manning.com/CPlusPlusConcurrencyinAction。

## 软件需求

为了直截了当地使用本书中的代码，你需要一个最新的C++编译器，它支持示例中使用的新的C++11语言特性（参见附录A），并且你需要C++标准线程库的副本。

在撰写本书的时候，g++是我所知道的唯一带有标准线程库实现的编译器，尽管Microsoft Visual Studio 2011预览版也包括了实现。线程库的g++实现最早是在g++ 4.3中引入了基本形式，并且在后来的版本中进行了扩展。g++ 4.3也引入了一些新C++11语言特性的初次支持；对新语言特性更多的支持在各个后续版本中。详情参见g++ C++11状态页面[1]。

Microsoft Visual Studio 2010提供了一些新C++11语言特性，例如右值引用和lambda

[1] GNU Compiler Collection C++0x/C++11 状态页面，http://gcc.gnu.org/projects/cxx0x.html。

函数，但并不带有线程库的实现。

作者的公司 Just Soft Solutions Ltd，出售为 Microsoft Visual Studio 2005、Microsoft Visual Studio 2008、Microsoft Visual Studio 2010 和 g++的各种版本[1]设计的 C++11 标准线程库的完整实现。这些实现已被用来测试本书中的示例。

Boost 线程库[2]提供了基于 C++标准线程库提案的 API，可以移植到许多平台。本书中的大部分示例可以通过审慎地将 `std::`替换成 `boost::`进行修改，并使用适当的 `#include` 指令，来与 Boost 线程库一起工作。在 Boost 线程库中有少数工具不受支持（比如 `std::async`）或者拥有不同的名称（比如 `boost::unique_future`）。

## 作者在线

购买 C++ Concurrency in Action 包括了免费访问由 Manning 出版社运营的私有网络论坛，在这里你可以对本书做出评论，提问技术问题，并且从作者和其他用户那里得到帮助。如果要访问此论坛并订阅它，可以访问网址 www.manning.com/CPlusPlusConcurrencyinAction。该页面提供了论坛的使用指南以及论坛上的行为规则。

Manning 对我们读者的承诺是提供一个场所，在这里每个读者之间以及读者和作者之间可以进行有意义的对话。就作者而言并没有承诺任何规定的参与量，作者对本书论坛的贡献只是义务的（且无偿的）。我们建议你试着向作者提问一些有挑战性的问题，以免他失去兴趣！

作者在线论坛以及过往讨论的归档，在本书在印期间都可以从出版社网站进行访问。

[1] C++标准线程库的 just::thread 实现，http://www.stdthread.co.uk。

[2] Boost C++库集合，http://www.boost.org。

# 资源

## 印刷资源

Cargill, Tom, "Exception Handling: A False Sense of Security,"in *C++ Report* 6, no. 9, (November-December 1994). Also available at http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html.

Hoare, C.A.R., *Communicating Sequential Processes* (Prentice Hall International, 1985), ISBN 0131532898. Also available at http://www.usingcsp.com/cspbook.pdf.

Michael, Maged M., "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes"in *PODC'02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (2002), ISBN 1-58113-485-1.

——. U.S. Patent and Trademark Office application 20040107227, "Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation."

Sutter, Herb, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* (Addison Wesley Professional, 1999), ISBN 0-201-61562-2.

———. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,"in *Dr. Dobb's Journal* 30, no. 3 (March 2005). Also available at http://www.gotw.ca/publications/concurrency-ddj.htm.

## 在线资源

Atomic Ptr Plus Project Home, http://atomic-ptr-plus.sourceforge.net/.

Boost C++ library collection, http://www.boost.org.

C++0x/C++11 Support in GCC, http://gcc.gnu.org/projects/cxx0x.html.

C++11—The Recently Approved New ISO C++ Standard, http://www.research.att.com/～bs/C++0xFAQ.html.

Erlang Programming Language, http://www.erlang.org/.

GNU General Public License, http://www.gnu.org/licenses/gpl.html.

Haskell Programming Language, http://www.haskell.org/.

IBM Statement of Non-Assertion of Named Patents Against OSS, http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf.

Intel Building Blocks for Open Source, http://threadingbuildingblocks.org/.

The just::thread Implementation of the C++ Standard Thread Library, http://www.stdthread.co.uk.

Message Passing Interface Forum, http://www.mpi-forum.org/.

Multithreading API for C++0X--A Layered Approach, C++ Standards Committee Paper N2094, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2094.html.

OpenMP, http://www.openmp.org/.

SETI@Home, http://setiathome.ssl.berkeley.edu/.

# 简要目录

# 目录

# 第 1 章　你好，C++并发世界

本章主要内容

- 何谓并发和多线程
- 为什么要在应用程序中使用并发和多线程
- C++并发支持的发展历程
- 一个简单的 C++多线程程序是什么样的

这是令 C++用户振奋的时刻。距 1998 年初始的 C++标准发布 13 年后，C++标准委员会给予程序语言和它的支持库一次重大的变革。新的 C++标准（也被称为 C++11 或 C++0x）于 2011 年发布并带来了很多的改变，使得 C++的应用更加容易并富有成效。

在 C++11 标准中一个最重要的新特性就是支持多线程程序。这是 C++标准第一次在语言中承认多线程应用的存在，并在库中为编写多线程应用程序提供组件。这将使得在不依赖平台相关扩展下编写多线程 C++程序成为可能，从而允许以有保证的行为来编写可移植的多线程代码。这也恰逢程序员寻求更多普遍的并发，特别是多线程程序，来提高应用程序的性能。

这本书讲述的就是 C++编程中对多线程并发的使用，以及相关的 C++语言特性和库工具。我会以解释并发和多线程的含义以及为什么要在应用程序中使用并发开始。在快速全方位地阐述为什么在应用程序中会不使用并发之后，我会对 C++中并发支持进行概述，并以一个简单的 C++并发实例结束这一章。具有开发多线程应用程序经验的读者可以跳过前面的小节。在随后几章将会涵盖更多广泛的例子，并且更深入地了解库工具。

本书最后附有对多线程与并发全部的 C++标准库工具的深入参考。

那么，什么是**并发**（**concurrency**）和**多线程**（**multithreading**）？

## 1.1 什么是并发

在最简单和最基本的层面，并发是指两个或更多独立的活动同时发生。并发在生活中随处可见。我们可以一边走路一边说话，也可以两只手同时做不同的动作，还有我们每个人都相互独立地过我们的生活——我在游泳的时候你可以看球赛，等等。

### 1.1.1 计算机系统中的并发

当我们提到计算机术语的“并发”，指的是在单个系统里同时执行多个独立的活动，而不是顺序地或是一个接一个地。这并不是一种新的现象，多任务操作系统通过任务切换允许一台计算机在同一时间运行多个应用程序已司空见惯多年，一些高端的多任务处理服务器实现并发控制的历史更久远。真正有新意的是增加计算机真正并行运行多任务的普遍性，而不只是给人这种错觉。

以前，大多数计算机都有一个处理器，具有单个处理单元或核心，至今许多台式机器仍是这样。这种计算机在某一时刻只可以真正执行一个任务，但它可以每秒切换任务许多次。通过做一点这个任务然后再做一点别的任务，看起来像是任务在并行发生。这就是**任务切换**（**task switching**）。我们仍然将这样的系统称为**并发**（**concurrency**），因为任务切换得太快，以至于无法分辨任务在何时会被暂挂而切换到另一个任务。任务切换给用户和应用程序本身造成了一种并发的假象。由于这只是并发的**假象**，当应用程序执行在单处理器任务切换环境下，与在真正的并发环境下执行相比，其行为还是有着微妙的不同。特别地，对内存模型不正确的假设（参见第 5 章）在这样的环境中可能不会出现。这将在第 10 章中作深入讨论。

包含多个处理器的计算机用于服务器和高性能计算任务已有多年，现在基于单个芯片上具有多于一个核心的处理器（多核心处理器）的计算机也成为越来越常见的台式机器。无论它们拥有多个处理器或一个多核处理器（或两者兼具），这些计算机能够真正的并行运行超过一个任务。我们才称之为**硬件并发**（**hardware concurrency**）。

图 1.1 显示了一个计算机处理恰好两个任务时的理想情景，每个任务被分为 10 个相等大小的块。在一个双核机器（具有两个处理核心）中，每个任务可以在各自的核心执行。在单核机器上做任务切换时，每个任务的块交织进行。但它们也隔开了一位（图中所示灰色分隔条的厚度大于双核机器的分隔条）。为了实现交替进行，该系统每次从一个任务切换到另一个时都得执行一次**上下文切换**（**context switch**），而这是需要时间的。为了执行上下文切换，操作系统必须为当前运行的任务保存 CPU 的状态和指令指针，算出要切换到哪

个任务，并为要切换到的任务重新加载处理器状态。然后 CPU 可能要将新任务的指令和数据的内存载入到缓存中，这可能会阻止 CPU 执行任何指令，造成进一步的延迟。

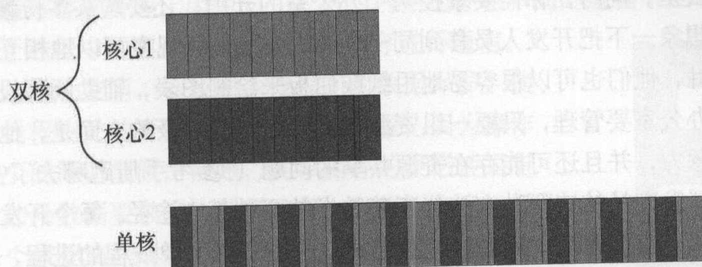



图 1.1　并发的两种方式：双核机器的并行执行对比单核机器的任务切换

尽管硬件并发的可用性在多处理器或多核系统上更显著，有些处理器却可以在一个核心上执行多个线程。要考虑的最重要的因素是硬件线程（**hardware threads**）的数量：即硬件可以真正并发运行多少独立的任务。即便是具有真正硬件并发的系统，也很容易有超过硬件可并行运行的任务要执行，所以在这些情况下任务切换仍将被使用。例如，在一个典型的台式计算机上可能会有几百个的任务在运行，执行后台操作，即使计算机在名义上是空闲的。正是任务切换使得这些后台任务可以运行，并使得你可以同时运行文字处理器、编译器、编辑器和 web 浏览器（或任何应用的组合）。图 1.2 显示了四个任务在一台双核机器上的任务切换，仍然是将任务整齐地划分为同等大小块的理想情况。实际上，许多因素造成了分割不均和调度不规则。这些因素中的一部分将涵盖在第 8 章中，那时我们再来看一看影响并行代码性能的因素。

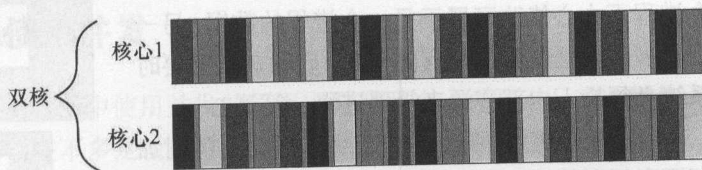



图 1.2　四个任务在两个核心之间的切换

所有的技术、功能和本书所涉及的类都可以使用，无论你的应用程序是在单核处理器还是多核处理器上运行，也不管是任务切换或是真正的硬件并发。但你可以想象，如何在你的应用程序中使用并发很大程度上取决于可用的硬件并发。这将在第 8 章中涵盖，在第 8 章我们具体研究 C++代码并行设计问题。

## 1.1.2　并发的途径

想象一下两个程序员一起做一个软件项目。如果你的开发人员在独立的办公室，它

们可以各自平静地工作，而不会互相干扰，并且他们各有自己的一套参考手册。然而，沟通起来就不那么直接了；不能转身然后互相交谈，他们必须用电话、电子邮件或走到对方的办公室。同时，你需要掌控两个办公室的开销，还要购买多份参考手册。

现在想象一下把开发人员移到同一间办公室。他们现在可以地相互交谈来讨论应用程序的设计，他们也可以很容易地用纸或白板来绘制图表，辅助阐释设计思路。你现在只有一个办公室要管理，只要一组资源就可以满足。消极的一面是，他们可能会发现难以集中注意力，并且还可能存在资源共享的问题（“参考手册跑哪去了？”）。

组织开发人员的这两种方法代表着并发的两种基本途径。每个开发人员代表一个线程，每个办公室代表一个处理器。第一种途径是有多个单线程的进程，这就类似让每个开发人员在他们自己的办公室，而第二种途径是在单一进程里有多个线程，这就类似在同一个办公室里有两个开发人员。你可以随意进行组合，并且拥有多个进程，其中一些是多线程的，一些是单线程的，但原理是一样的。让我们在一个应用程序中简要地看一看这两种途径。

### 1. 多进程并发

在一个应用程序中使用并发的第一种方法，是将应用程序分为多个、独立的、单线程的进程，它们运行在同一时刻，就像你可以同时进行网页浏览和文字处理。这些独立的进程可以通过所有常规的进程间通信渠道互相传递信息（信号、套接字、文件、管道等），如图1.3所示。有一个缺点是这种进程之间的通信通常设置复杂，或是速度较慢，或两者兼备，因为操作系统通常在进程间提供了大量的保护，以避免一个进程不小心修改了属于另一个进程的数据。另一个缺点是运行多个进程所需的固有的开销：启动进程需要时间，操作系统必须投入内部资源来管理进程， 等等。

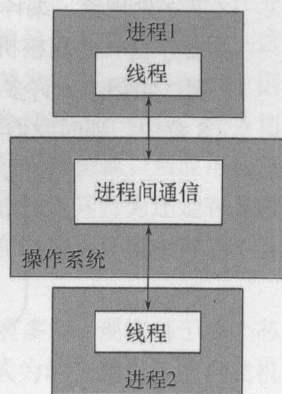



图1.3 一对并发运行的进程之间的通信

当然，也并不全是缺点：操作系统在线程间提供的附加保护操作和更高级别的通信机制，意味着可以比线程更容易地编写安全的并发代码。事实上，类似于为Erlang编程语言提供的环境， 可使用进程作为重大作用并发的基本构造块。

使用独立的进程实现并发还有一个额外的优势——你可以通过网络连接的不同的机器上运行独立的进程。虽然这增加了通信成本，但在一个精心设计的系统上，它可能是一个提高并行可用行和提高性能的低成本方法。

### 2. 多线程并发

并发的另一个途径是在单个进程中运行多个线程。线程很像轻量级的进程：每个线程相互独立运行，且每个线程可以运行不同的指令序列。但进程中的所有线程都共享相

同的地址空间，并且从所有线程中访问到大部分数据——全局变量仍然是全局的，指针、对象的引用或数据可以在线程之间传递。虽然通常可以在进程之间共享内存，但这难以建立并且通常难以管理，因为同一数据的内存地址在不同的进程中也不尽相同。图 1.4 显示了一个进程中的两个线程通过共享内存进行通信。

共享的地址空间，以及缺少线程间的数据保护，使得使用多线程相关的开销远小于使用多进程，因为操作系统有更少的簿记要做。但是，共享内存的灵活性是有代价的：如果数据要被多个线程访问，那么程序员必须确保当每个线程访问时所看到的数据是一致的。线程间数据共享可能会遇到的问题、所使用的工具以及为了避免问题而要遵循的准则在本书中都有涉及，特别是在第 3、4、5 和 8 章中。这些问题并非不能克服，只要在编写代码时适当地注意即可，但这却意味着必须对线程之间的通信作大量的思考。

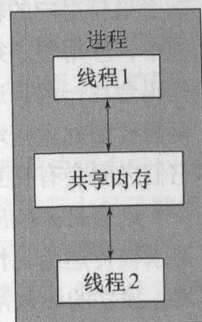



图 1.4 同一进程中的一对并发运行的线程之间的通信

相比于启动多个单线程进程并在其间进行通信，启动单一进程中的多线程并在其间进行通信的开销更低，这意味着若不考虑共享内存可能会带来的潜在问题，它是包括 C++在内的主流语言更青睐的并发途径。此外，C++标准没有为进程间通信提供任何原生支持，所以使用多进程的应用程序将不得不依赖平台相关的 API 来实现。因此，本书专门关注使用多线程的并发，并且之后提到并发均是假定通过使用多线程来实现的。

明确了什么是并发后，现在让我们来看看为什么要在应用程序中使用并发。

## 1.2 为什么使用并发

在应用程序中使用并发的原因主要有两个：关注点分离和性能。事实上，我甚至可以说它们差不多是使用并发的唯一原因，当你观察得足够仔细时，一切其他因素都可以归结到这两者之一（或者可能是二者兼有，当然，除了像“我愿意”这样的原因之外）。

### 1.2.1 为了划分关注点而使用并发

在编写软件时，划分关注点总是个好主意。通过将相关的代码放在一起并将无关的代码分开，这种方法可以使你的程序更容易理解和测试，从而减少出错的可能性。你可以使用并发来分隔不同的功能区域，即使在这些不同功能区域的操作需要在同一时刻发生的情况下。如果不显式地使用并发，你要么被迫编写任务切换框架，要么在操作中主动地调用不相关的一段代码。

考虑一类带有用户界面的密集处理型应用程序，例如为台式计算机提供的DVD播放程序。这样一个应用程序基本上具备两套职能：它不仅要从光盘中读取数据，解码图像和声音，并把它们及时输出至视频和音频硬件，从而实现DVD的无错播放；它还要接受来自用户的输入，例如当用户单击暂停或返回菜单甚至退出按键的情况。在单个线程中，应用程序须在回放期间定期检查用户的输入，于是将DVD回放代码和用户界面代码合在一起。通过使用多线程来分隔这些关注点，用户界面代码和DVD回放代码不再需要如此紧密地交织在一起。一个线程可以处理用户界面，另一个处理DVD回放，它们之间会有交互，例如用户点击暂停，但现在这些交互直接与眼前的任务有关。

这会带来响应性的错觉，因为用户界面线程通常可以立即响应用户的请求，即使在请求被传达给工作的线程，响应为简单地显示正忙的光标或请等待的消息的情况。类似地，独立的线程常被用于运行必须在后台连续运行的任务，例如在桌面搜索程序中监视文件系统的变化。以这种方式使用线程一般会使每个线程的逻辑更加简单，因为它们之间的交互可以被限制为清晰可辨的点，而不是到处散播不同任务的逻辑。

在这种情况下，线程的数量与CPU可用内核的数量无关，因为对线程的划分是基于概念上的设计而不是试图增加吞吐量。

### 1.2.2 为了性能而使用并发

多处理器系统已经存在了几十年，但直到最近，他们几乎只能在超级计算机、大型机和大型服务器系统中才能看到。然而芯片制造商越来越倾向于多核芯片的设计，即在单个芯片上集成2、4、16或更多的处理器，从而达到比单核心更好的性能。因此，多核台式计算机，甚至多核嵌入式设备，现在越来越普遍。这些计算机的计算能力的提高不是源自使单一任务运行的更快，而是源自并行运行多个任务。在过去，程序员曾坐等他们的程序随着处理器的更新换代而变得更快，无需他们这边做出任何努力。但是现在，就像Herb Sutter所说的，“免费的午餐结束了[1]”。**如果软件想要利用日益增长的计算能力，它必须设计为并发运行多个任务。**程序员因此必须留意，而且那些迄今都忽略并发的人们必须注意它并将其加入他们的工具箱中。

有两种方式为了性能使用并发。首先，也是最明显的，是将一个单个任务分成几部分且各自并行运行，从而降低总运行时间，这就是**任务并行（task parallelism）**。虽然这听起来很直观，但它可以是一个相当复杂的过程，因为在各个部分之间可能存在很多的依赖。区别可能是在过程方面——一个线程执行算法的一部分而另一个线程执行算法的另一部分——或是在数据方面——每个线程在不同的数据部分上执行相同的操作。后一种方法被称为**数据并行（data parallelism）**。

[1] *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, Herb Sutter, Dr. Dobb's Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm

容易受这种并行影响的算法常被称为**易并行**（**embarrassingly parallel**）。抛开你可能会尴尬地面对的很容易并行化的代码这一含义，这是一件好事情。我曾遇到过的关于此算法的别的术语是**自然并行**（**naturally parallel**）和**便利并发**（**conveniently concurrent**）。易并行算法具有良好的可扩展特性——随着可用硬件线程数量的提升，算法的并行性可以随之增加与之匹配。这样的一个算法是谚语“人多力量大”的完美体现。对于非易并行算法的那一部分，你可以将算法划分为一个固定（因而不可扩展）数量的并行任务。在线程之间划分任务的技巧涵盖在第 8 章中。

使用并发来提升性能的第二种方法是使用可用的并行方式来解决更大的问题。与其同时处理一个文件，不如酌情处理 2 个或 10 个或 20 个。虽然这实际上只是**数据并行**的一种应用，通过对多组数据同时执行相同的操作，但还是有不同的重点。处理一个数据块仍然需要同样的时间，但在相同的时间内却可以处理更多的数据。当然，这种方法也存在限制，且并非在所有情况下都是有益的，但是这种方法所带来的吞吐量提升可以让一些新玩意变得可能。例如，如果图片的各部分可以并行处理，就能提高视频处理的分辨率。

## 1.2.3 什么时候不使用并发

知道**何时不**使用并发与知道**何时要**使用它同等重要。基本上，不使用并发的唯一原因就是在收益比不上成本的时候。使用并发的代码在很多情况下难以理解，因此编写和维护的多线程代码就有直接的脑力成本，同时额外的复杂性也可能导致更多的错误。除非潜在的性能增益足够大或关注点分离得足够清晰，能抵消确保其正确所需的额外的开发时间以及与维护多线程代码相关的额外成本，否则不要使用并发。

同样地，性能增益可能不会如预期的那么大。在启动线程时存在固有的开销，因为操作系统必须分配相关的内核资源和堆栈空间，然后将新线程加入调度器中，所有这一切都要占用时间。如果在线程上运行的任务完成得很快，那么任务实际上占据的时间与启动线程的开销时间相比就显得微不足道，可能会导致应用程序的整体性能还不如通过产生线程直接执行该任务。

此外，线程是有限的资源。如果让太多的线程同时运行，则会消耗操作系统资源，并且使得操作系统整体上运行得更缓慢。不仅如此，运行太多的线程会耗尽进程的可用内存或地址空间，因为每个线程都需要一个独立的堆栈空间。对于一个可用地址空间限制为 4GB 的扁平架构的 32 位进程来说，这尤其是个问题。如果每个线程都有一个 1MB 的堆栈（对于很多系统来说是典型的），那么 4096 个线程将会用尽所有地址空间，不再为代码、静态数据或者堆数据留有空间。虽然 64 位（或者更大）的系统不存在这种直接的地址空间限制，它们仍然只具备有限的资源：如果你运行太多的线程，最终会导致问题。尽管线程池（参见第 9 章）可以用来限制线程的数量，但这并不是灵丹妙药，它

们也有自己的问题。

如果客户端/服务器应用程序的服务器端为每一个链接启动一个独立的线程，对于少量的链接是可以正常工作的，但当同样的技术用于需要处理大量链接的高需求服务器时，就会因为启动太多线程而迅速耗尽系统资源。在这种场景下，谨慎地使用线程池可以提供优化的性能（参见第9章）。

最后，运行越多的线程，操作系统就需要做越多的上下文切换。每个上下文切换都需要耗费本可以花在有价值工作上的时间，所以在某些时候，增加一个额外的线程实际上会降低而不是提高应用程序的整体性能。为此，如果你试图得到系统的最佳性能，考虑可用的硬件并发（或缺乏之）并调整运行线程的数量是必需的。

为了性能优化而使用并发就像所有其他优化策略一样，它拥有极大提高应用程序性能的潜力，但它也可能使代码复杂化，使其更难理解和更容易出错。因此，只有对应用程序中的那些具有显著增益潜力的性能关键部分才值得这样做。当然，如果性能收益的潜力仅次于设计清晰或关注点分离，可能也值得使用多线程设计。

假设你已经决定确实要在应用程序中使用并发，无论是为了性能、关注点分离，或是因为“多线程星期一”，对于C++程序员来说意味着什么？

## 1.3 在C++中使用并发和多线程

通过多线程为并发提供标准化的支持对 C++来说是新鲜事物。只有在即将到来的C++11标准中，你才能不依赖平台相关的扩展来编写多线程代码。为了理解新版本C++线程库中众多规则背后的基本原理，了解其历史是很重要的。

### 1.3.1 C++多线程历程

1998C++标准版不承认线程的存在，并且各种语言要素的操作效果都以顺序抽象机的形式编写。不仅如此，内存模型也没有被正式定义，所以对于1998 C++标准，你没办法在缺少编译器相关扩展的情况下编写多线程应用程序。

当然，编译器供应商可以自由地向语言添加扩展，并且针对多线程的 C API 的流行——例如在 POSIX C 和 Microsoft Windows API 中的那些——导致很多 C++编译器供应商通过各种平台相关的扩展来支持多线程。这种编译器支持普遍地受限于只允许使用该平台相应的 C API 以及确保该 C++运行时库（例如异常处理机制的代码）在多线程存在的情况下运行。尽管极少有编译器供应商提供了一个正式的多线程感知内存模型，但编译器和处理器的实际表现也已经足够好，以至于大量的多线程的C++程序已被编写出来。

由于不满足于使用平台相关的 C API 来处理多线程，C++程序员曾期望他们的类库

提供面向对象的多线程工具。像 MFC 这样的应用程序框架，以及像 Boost 和 ACE 这样的C++通用类库曾积累了多套 C++类，封装了下层的平台相关 API 并提供高级的多线程工具以简化任务。各类库的具体细节，特别是在启动新线程的方面，存在很大差异，但是这些类的总体构造存在很多共通之处。有一个为许多 C++类库共有的，同时也是为程序员提供很大便利的特别重要的设计，就是带锁的资源获得即初始化（**RAII, Resource Acquisition Is Initialization**）的习惯用法，来确保当退出相关作用域的时候互斥元被解锁。

许多情况下，现有的 C++编译器所提供的多线程支持，例如 Boost 和 ACE，综合了平台相关API以及平台无关类库的可用性，为编写多线程C++代码提供一个坚实的基础，也因此大约有数百万行 C++代码作为多线程应用程序的一部分而被编写出来。但缺乏标准的支持，意味着存在缺少线程感知内存模型从而导致问题的场合，特别是对于那些试图通过使用处理器硬件能力来获取更高性能，或是编写跨平台代码，但是在不同平台之间编译器的实际表现存在差异。

## 1.3.2 新标准中的并发支持

所有这些都随着新的 C++11 标准的发布而改变了。不仅有了一个全新的线程感知内存模型，C++标准库也被扩展了，包含了用于管理线程（参见第 2 章）、保护共享数据（参见第 3 章）、线程间同步操作（参见第 4 章）以及低级原子操作（参见第 5 章）的各个类。

新的 C++线程库很大程度上基于之前通过使用上文提到的 C++类库而积累的经验。特别地，Boost 线程库被用作新类库所基于的主要模型，很多类与 Boost 中的对应者共享命名和结构。在新标准演进的过程中，这是个双向流动，Boost 线程库也改变了自己，以便在多个方面匹配 C++标准，因此从 Boost 迁移过来的用户将会发现自己非常适应。

正如本章开篇提到的那样，对并发的支持仅仅是新 C++标准的变化之一，此外还存在很多对于编程语言自身的改善，可以使得程序员们的工作更便捷。这些内容虽然不在本书的论述范围之内，但是其中的一些变化对于线程库本身及其使用方式已经形成了直接的冲击。附录 A 对这些语言特性做了简要的介绍。

C++中对原子操作的直接支持，允许程序员编写具有确定语义的高效代码，而无需平台相关的汇编语言。这对于那些试图编写高效的、可移植代码的程序员们来说是一个真正的福利。不仅有编译器可以搞定平台的具体内容，还可以编写优化器来考虑操作的语义，从而让程序作为一个整体得到更好的优化。

## 1.3.3 C++线程库的效率

对于 C++整体以及包含低级工具的 C++类——特别是在新版 C++线程库里的那些，

参与高性能计算的开发者常常关注的一点就是效率。如果你正寻求极致的性能，那么理解与直接使用底层的低级工具相比，使用高级工具所带来的实现成本，是很重要的。这个成本就是抽象惩罚（**abstraction penalty**）。

C++标准委员会在整体设计 C++标准库以及专门设计标准 C++线程库的时候，就已经十分注重这一点了。其设计的目标之一就是在提供相同的工具时，通过直接使用低级 API 就几乎或完全得不到任何好处。因此该类库被设计为在大部分平台上都能高效实现(带有非常低的抽象惩罚)。

C++标准委员会的另一个目标，是确保 C++能提供足够的低级工具给那些希望与硬件工作得更紧密的程序员，以获取终极性能。为了达到这个目的，伴随着新的内存模型，出现了一个全面的原子操作库，用于直接控制单个位、字节、线程间同步以及所有变化的可见性。这些原子类型和相应的操作现在可以在很多地方加以使用，而这些地方以前通常被开发者选择下放到平台相关的汇编语言中。使用了新的标准类型和操作的代码因而具有更佳的可移植性，并且更易于维护。

C++标准库也提供了更高级别的抽象和工具，它们使得编写多线程代码更简单和不易出错。有时候运用这些工具确实会带来性能成本，因为必须执行额外的代码。但是这种性能成本并不一定意味着更高的抽象惩罚，总体来看，这种性能成本并不比通过手工编写等效的函数而招致的成本更高，同时编译器可能会很好地内联大部分额外的代码。

在某些情况下，高级工具提供超出特定使用需求的额外功能。在大部分情况下这都不是问题，你没有为你不使用的那部分买单。在罕见的情况下，这些未使用的功能会影响其他代码的性能。如果你更看重程序的性能，且代价过高，你可能最好是通过较低级别的工具来手工实现需要的功能。在绝大多数情况下，额外增加的复杂性和出错的几率远大于小小的性能提升所带来的潜在收益。即使有证据确实表明瓶颈出现在 C++标准库的工具中，这也可能归咎于低劣的应用程序设计而非低劣的类库实现。例如，如果过多的线程竞争一个互斥元，这将会显著影响性能。与其试图在互斥操作上花掉一点点的时间，还不如重新构造应用程序以减少互斥元上的竞争来得划算。设计应用程序以减少竞争会在第 8 章中加以阐述。

在非常罕见的情况下，C++标准库不提供所需的性能或行为，这时则有必要运用特定的平台相关的工具。

### 1.3.4 平台相关的工具

虽然 C++线程库为多线程和并发处理提供了颇为全面的工具，但是在所有的平台上，都会有些额外的平台相关工具。为了能方便地访问那些工具而又不用放弃使用标准 C++线程库带来的好处，C++线程库中的类型可以提供一个 `native_handle()`成员函数，允许通过使用平台相关 API 直接操作底层实现。就其本质而言，任何使用

native_handle()执行的操作是完全依赖于平台的，这也超出了本书（同时也是标准C++库本身）的范围。

当然，在考虑使用平台相关的工具之前，明白标准库能够提供什么是很重要的，那么让我们通过一个例子来开始。

## 1.4 开始入门

好，现在你有一个很棒的与 C++11 兼容的编译器。接下来呢？一个多线程 C++程序是什么样子的？它看上去和其他所有 C++程序一样，通常是变量、类以及函数的组合。唯一真正的区别在于某些函数可以并发运行，所以你需要确保共享数据的并发访问是安全的，详见第 3 章。当然，为了并发地运行函数，必须使用特定的函数以及对象来管理各个线程。

### 你好，并发世界

让我们从一个经典的例子开始：一个打印“Hello World.”的程序。一个非常简单的在单线程中运行的 Hello, World 程序如下所示，当我们谈到多线程时，它可以作为一个基准。

```
#include <iostream>

int main()
{
    std::cout<<"Hello World\n";
}
```

清单 1.1 这个程序所做的一切就是将“Hello World”写进标准输出流。让我们将它与下面清单所示的简单的 Hello, Concurrent World 程序做个比较，它启动了一个独立的线程来显示这个信息。

**清单 1.1　一个简单的 Hello,Concurrent World 程序**

```
#include <iostream>
#include <thread>                        <-❶

void hello()                                 <-❷
{
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello);            <-❸
    t.join();                                <-❹
}
```

第一个区别是增加了#include<thread>❶。在标准 C++库中对多线程支持的声明在新的头文件中，用于管理线程的函数和类在<thread>中声明，而那些保护共享数据的函数和类在其他头文件中声明。

其次，写信息的代码被移动到了一个独立的函数中❷。这是因为每个线程都必须具有一个初始函数（**initial function**），新线程的执行在这里开始。对于应用程序来说，初始线程是main()，但是对于所有其他线程，这在std::thread对象的构造函数中指定——在本例中，被命名为t❸ 的std::thread对象拥有新函数hello()作为其初始函数。

下一个区别，与直接写入标准输出或是从main()调用hello()不同，该程序启动了一个全新的线程来实现，将线程数量一分为二——初始线程始于main()而新线程始于hello()。

在新的线程启动之后❸，初始线程继续执行。如果它不等待新线程结束，它就将自顾自地继续运行到main()的结束，从而结束程序——有可能发生在新线程有机会运行之前。这就是为什么在❹ 这里调用join()的原因——详见第 2 章，这会导致调用线程（在main()中）等待与std::thread对象相关联的线程，即这个例子中的t。

如果这看起来像是仅仅为了将一条信息写入标准输出而做了大量的工作，那么它确实如此——正如上文 1.2.3 节所描述的，一般来说并不值得为了如此简单的任务而使用多线程，尤其是如果在这期间初始线程无所事事。在本书后面的内容中，我们将通过实例来展示在哪些情景下使用多线程可以获得明确的收益。

## 1.5 小结

在本章中，我提及了并发与多线程的含义以及在你的应用程序中为什么会选择使用（或不使用）它。我还提及了多线程在 C++中的发展历程，从 1998 标准中完全缺乏支持，经历了各种平台相关的扩展，再到新的 C++11 标准中具有合适的多线程支持。该支持到来的正是时候，它使得程序员们可以利用伴随新的 CPU 而带来的更加强大的硬件并发，因为芯片制造商选择了以多核心的形式使得更多任务可以同时执行的方式来增加处理能力，而不是增加单个核心的执行速度。

在 1.4 节中的示例中展示了 C++标准库中的类和函数有多么的简单。在 C++中，使用多线程本身并不复杂，复杂的是如何设计代码以实现其预期的行为。

在尝试了 1.4 节的示例之后，是时候看看更多实质性的内容了。在第 2 章中，我们将看一看用于管理线程的类和函数。

# 第 2 章　管理线程

本章主要内容

- 启动线程，以及各种让代码在新线程上运行的方法
- 等待线程完成并让它自动运行
- 唯一地标识线程

那么，你已经下决心为你的应用程序使用并发了。特别地，你决定了使用多线程。接下来呢？如何启动这些线程，怎样检查它们已完成，怎样监视它们呢？C++标准库让大多数线程管理任务变得相对简单，通过与给定线程相关联的 std::thread 对象就可以管理所有事情，如你将要看到的那样。对于那些并不直观的任务，标准库也提供了从基本构建块进行按需构建的可扩展性。

在本章中，我将从基础开始阐述。启动一个线程，等待它完成，或是在后台运行它。接下来我们将看一看在线程函数启动时向其传递额外的参数，以及如何将线程的所有权从一个 std::thread 对象转移到另一个。最后，我们会看一看选择所使用的线程数量，以及标识特定的线程。

## 2.1　基本线程管理

每个 C++程序都拥有至少一个线程，它是由 C++在运行时启动的，该线程运行着

main()函数。你的程序可以继续启动具有其他函数作为入口的线程。然后，这些线程连同初始线程一起，并发运行。正如程序会在main()函数返回时退出那样，当指定的入口函数返回时，该线程就会退出。如你所见，如果你有线程对应的std::thread对象，你就可以等待它完成。但首先你得启动它，所以让我们来看一看启动线程。

### 2.1.1 启动线程

如同你在第 1 章中所看到的，线程是通过构造std::thread对象来开始的，该对象指定了线程上要运行的任务。在最简单的情况下，该任务仅仅是一个普普通通的返回void且不接受参数的函数。这个函数在自己的线程上运行，直到返回，然后线程停止。但从另一个极端看，该任务可能是一个接受额外参数的函数对象，当它运行时，会执行一系列由某种消息机制所指定的相互独立的操作，并且只有当线程再次通过某种消息机制接收到信号时才会停止。无论线程将要做什么或是从哪里启动，使用 C++线程库来开始一个线程总归是要构造一个std::thread对象。

```
void do_some_work();
std::thread my_thread(do_some_work);
```

就是这么简单。当然，你必须确保引入了<thread>头文件，从而编译器可以找到std::thread类的定义。与许多 C++标准库相似，std::thread可以与任何可调用（**callable**）类型一同工作，所以你可以将一个带有函数调用操作符的类的实例传递给std::thread的构造函数来进行代替。

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
};
background_task f;
std::thread my_thread(f);
```

在这种情况下，所提供的函数对象被**复制**（**copied**）到属于新创建的执行线程的存储器中，并从那里调用。因此重要的是，副本与原版有着等效的行为，否则结果可能会与预期不符。

当给线程构造函数传递一个函数对象时要考虑的一件事是避免所谓的“C++的最棘手的解析”。如果你传递一个临时的且未命名的变量，那么其语法可能与函数声明一样，在这种情况下，编译器会将其解释成如下这样，而非对象定义。例如，

```
std::thread my_thread(background_task());
```

声明了函数 my_thread，它接受单个参数（参数类型是指向不接受参数同时返回 background_task 对象的函数的指针），并返回 std::thread 对象，而不是启动一个新线程。你可以像前面说的那样通过命名函数对象来避免这种情况，通过使用一组额外的括号，或使用新的统一初始化语法，例如，

```
std::thread my_thread((background_task()));   ❶
std::thread my_thread{background_task()};     ❷
```

在第一个例子❶中，额外的括号避免其解释为函数声明，从而让 my_thread 被声明为 std::thread 类型的变量。第二个例子❷使用新的统一初始化语法，用大括号而不是括号，同样也是声明一个变量。

有一种避免了此问题的可调用对象类型，就是 **lambda 表达式（lambda expression）**。这是 C++11 中的一项新功能，其基本功能是允许你编写一个局部函数，并可能捕捉一些局部变量，同时避免传递额外参数的需求（参见 2.2 节）。有关 lambda 表达式的详情，请参阅附录 A 中 A.5 节。前面的例子可以用 lambda 表达式编写如下。

```
std::thread my_thread([](
    do_something();
    do_something_else();
});
```

一旦开始了线程，你需要显式地决定是要等待它完成（通过结合它——参见 2.1.2 节），还是让它自行运行（通过分离它——参见 2.1.3 节）。如果你在 std::thread 对象被销毁前未作决定，那么你的程序会被终止（std::thread 的析构函数调用 std::terminate()）。因此，即便在异常存在的情况下，确保线程正确地结合或是分离都是你的当务之急。请参见 2.1.3 节中处理这一场景的技巧。需要注意的是，你只需要在 std::thread 对象被销毁之前做出这个决定即可——线程本身可能在你结合或分离它之前早就已经结束了，而且如果你分离它，那么该线程可能在 std::thread 对象被销毁后很久都还在运行。

如果你不等待线程完成，那么你需要确保通过该线程访问的数据是有效的，直到该线程完成为止。这并不是新问题——即使是在单线程代码中，在对象被销毁后还访问它也是未定义的行为——但线程的使用提供了遇到这种生命周期问题的额外机会。

你可能遇到这样问题的一种情况是，当线程函数持有局部变量的指针或引用，且当函数退出的时候线程尚未完成时，清单 2.1 展示的就是这样一个场景的例子。

**清单 2.1　当线程仍然访问局部变量时返回的函数**

```
struct func
{
    int& i;

    func(int& i_):i(i_){}
```

```
    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);          ← ❶ 对悬空引用可能的访问
        }
    }
};

void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();               ← ❷ 不等待线程完成
}                                     ← ❸ 新的线程可能仍在运行
```

在这种情况下，当oops退出❸时与my_thread相关联的新线程可能仍然在运行，因为通过调用detach()❷你已经显式地决定不等待它。如果线程仍在运行，则在下次调用do_something(i)❶时就会访问一个已被销毁的变量。这就像普通的单线程代码那样——允许对局部变量的指针或引用持续到函数退出之后绝不是一个好主意——但对于多线程代码更容易犯这样的错误，因为当它发生的时候，并不一定是显而易见的。

一个常见的处理这种情况的方式是使线程函数自包含，并且把数据复制（copy）到该线程中而不是共享数据。如果你为线程函数使用了一个可调用对象，该对象本身被复制到该线程中，那么原始对象就可以立即被销毁。但是你仍然需要警惕包含有指针或引用的对象，就像上面的清单2.1那样。特别地，在一个访问局部变量的函数中创建线程是个糟糕的主意，除非能保证线程在函数退出前完成。

另外，通过结合（**joining**）线程，你可以确保在函数退出前，该线程执行完毕。

## 2.1.2 等待线程完成

如果你需要等待线程完成，你可以通过在相关联的std::thread实例上调用join()来实现。在清单2.1的情况下，把函数体右括号前对my_thread.detach()的调用替换为调用my_thread.join()，就将足以确保在函数退出之前，即局部变量被销毁之前，该线程就已结束。在这种情况下，就意味着在独立的线程上运行函数是没什么意义的，因为第一个线程在此期间将做不了任何有用的事情，但在实际的代码当中，初始线程可能要么有自己的工作去做，要么是在等待所有线程完成之前就要启动多个线程来做有用的工作。

join()很简单也很暴力——你要么等待一个线程完成要么就不等。如果你需要对等待线程进行更细粒度的控制，比如检查线程是否完成，或只是在一段特定的时间内进行等待，那么就必须使用替代机制，例如条件变量和future，我们将在第4章中提到。调用join()的行为也会清理所有与该线程相关联的存储器，这样std::thread对象

不再与现已完成的线程相关联，它也不与任何线程相关联。这就意味着，你只能对一个给定的线程调用一次 join()，一旦你调用了 join()，此 std::thread 对象不再是可连接的，并且 joinable() 将返回 false。

### 2.1.3 在异常环境下的等待

如前所述，你要确保在 std::thread 对象被销毁前已调用 join() 或 detach() 函数。如果要分离线程，通常在线程启动后就可以立即调用 detach()，所以这不是个问题。但是如果打算等待该线程，就需要仔细地选择在代码的哪个位置调用 join()。这意味着，如果在线程开始之后但又是在调用 join() 之前引发了异常，对 join() 的调用就容易被跳过。

为了避免应用程序在引发异常的时候被终止，你需要在这种情况决定要做什么。一般来说，如果你打算在非异常的情况下调用 join()，你还需要在存在异常时调用 join()，以避免意外的生命周期问题。清单 2.2 展示了这样的简单代码。

**清单 2.2　等待线程结束**

```
struct func;                          <-- 参见清单 2.1 中的
                                          定义
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    try
    {
        do_something_in_current_thread();
    }
    catch(...)
    {
        t.join();        <-- ❶
        throw;
    }
    t.join();            <-- ❷
}
```

清单 2.2 中的代码使用了 try/catch 块，以确保访问局部状态的线程在函数退出前结束，无论函数是正常退出❷还是异常❶中断。使用 try/catch 块很啰嗦，而且容易将作用域弄乱，所以并不是一个理想的方案。如果确保线程必须在函数退出前完成是很重要的——无论是因为它具有对其他局部变量的引用还是任何其他原因——那么确保这是所有可能的退出路径的情况是很重要的，无论正常还是异常，并且希望提供一个这样做的简单明了的机制。

这样做的方法之一是使用标准的资源获取即初始化（**RAII**）惯用语法，并提供一个

类，在它的析构函数中进行 join()，正如清单 2.3 的代码。看看它是如何简化函数 f() 的。

**清单 2.3 使用 RAII 等待线程完成**

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_):
        t(t_)
    {}
    ~thread_guard()
    {
        if(t.joinable())                          <-❶
        {
            t.join();                             <-❷
        }
    }
    thread_guard(thread_guard const&)=delete;              <-❸
    thread_guard& operator=(thread_guard const&)=delete;
};

struct func;                                <- 参见清单 2.1 中的
                                               定义
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    thread_guard g(t);

    do_something_in_current_thread();
}                                           <-❹
```

在当前线程的执行到达 f 末尾❹时，局部对象会按照构造函数的逆序被销毁。因此，thread_guard 对象 g 首先被销毁，并且析构函数❷中线程被结合。即便是当函数因 do_something_in_current_thread 引发异常而退出的情况下也会发生。

在清单 2.3 中的析构函数在调用 join()❷前首先测试 thread_guard 的析构函数是不是 joinable()❶的。这很重要，因为对于一个给定的执行线程 join() 只能被调用一次，所以如果线程已经被结合，这样做就是错误的。

拷贝构造函数和拷贝赋值运算符被标记=delete❸，以确保他们不会由编译器自动提供。复制或赋值这样一个对象可能是危险的，因为它可能比它要结合的线程的作用域存在得更久。通过将它们声明为已删除的，任何复制 thread_guard 对象的企图都将产生编译错误。参见附录 A 中 A.2 节，了解更多关于已删除的函数。

如果无需等待线程完成，可以通过分离（**detaching**）它来避免这种异常安全问题。这打破了线程与 std::thread 对象的联系并确保当 std::thread 对象被销毁时

std::terminate()不会被调用，即使线程仍在后台运行。

### 2.1.4 在后台运行线程

在 std::thread 对象上调用 detach()会把线程丢在后台运行，也没有直接的方法与之通信。也不再可能等待该线程完成；如果一个线程成为分离的，获取一个引用它的 std::thread 对象也是不可能的，所以它也不再能够被结合。分离的线程确实是在后台运行；所有权和控制权被转交给 C++运行时库，以确保与线程相关联的资源在线程退出后能够被正确地回收。

参照 UNIX 的**守护进程**（**daemon process**）概念，被分离的线程通常被称为**守护线程**（**daemon threads**），它们无需任何显式的用户界面，而运行在后台。这样的线程通常是长时间运行的，它们可能在应用程序的几乎整个生命周期中都在运行，执行后台任务，例如监控文件系统、清除对象缓存中的未使用项或是优化数据结构。在另一个极端，有另一种鉴别线程何时完成的机制，或者线程被用作“即用即忘”任务，在这里使用分离线程也是有意义的。

如你在 2.1.2 节中已经看到的，你通过调用 std::thread 对象的 detach()的成员函数来分离线程。在调用完成后，std::thread 对象不再与执行的实际线程相关联，同时也不能够被加入。

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());
```

为了从一个 std::thread 对象中分离线程，必须有一个线程供分离。你不能在一个没有与执行线程相关联的 std::thread 对象上调用 detach()。这对于 join()也是同样的要求，你可以用完全相同的方法进行检查——你只能在 t.joinable()返回 true 的时候，为一个 std::thread 对象 t 调用 t.detach()。

考虑一个类似于字处理器的应用程序，它可以一次编辑多个文档。有许多种方法在 UI 级别和内部来处理这个问题。有一种现在看起来越来越普遍的方式，是具有多个相互独立的顶层窗口，与正在编辑的文档一一对应。尽管这些窗口看起来完全独立，各自拥有自己的菜单等，但它们是在同一个应用程序的实例上运行的。一种在内部处理这个问题的方式是在其自己的线程中运行各自的文档编辑窗口；每个线程都运行相同的代码，但拥有与被编辑文档相关的不同的数据以及相应的窗口属性。打开一个新的文档就需要启动一个新的线程。处理请求的线程并不在乎等待其他的线程完成，因为它在一个不相关的文件上工作，所以运行分离的线程就成为了首选。

清单 2.4 展示了这种方法的简单的代码大纲。

**清单 2.4 分离线程以处理其他文档**

```
void edit_document(std::string const& filename)
{
    open_document_and_display_gui(filename);
    while(!done_editing())
    {
        user_command cmd=get_user_input();
        if(cmd.type==open_new_document)
        {
            std::string const new_name=get_filename_from_user();
            std::thread t(edit_document,new_name);     ❶
            t.detach();     ❷
        }
        else
        {
            process_user_input(cmd);
        }
    }
}
```

如果用户选择打开一个新的文档，它会提示其有文档要打开，启动新线程来打开该文档❶，然后分离它❷。因为新的线程与当前线程做着同样的操作，只是文件不同，你可以用新选定的文件名作为参数，重用同一个函数（edit_document）。

这个例子还展示了一个案例，它有助于传递参数给用来启动线程的函数：并非仅仅将函数名传递给 std::thread 构造函数❶，你还可以传递文件名参数。虽然也有其他机制能够做到这一点，例如使用具有成员数据的函数对象取代普通的带有参数的函数，但线程库提供了一个简单方法来实现之。

## 2.2 传递参数给线程函数

如清单 2.4 中所示，传递参数给可调用对象或函数，基本上就是简单地将额外的参数传递给 std::thread 构造函数。但重要的是，参数会以默认的方式被复制（**copied**）到内部存储空间，在那里新创建的执行线程可以访问它们，即便函数中的相应参数期待着引用。这里有一个简单的例子。

```
void f(int i,std::string const& s);
std::thread t(f,3,"hello");
```

这里创建一个新的与 t 相关联的执行线程，称为 f(3, "hello")。注意即使 f 接受 std::string 作为第二个参数，字符串字面值仅在新线程的上下文中才会作为 char const*传送，并转换为 std::string。尤其重要的是当提供的参数是一个自动变量的指针时，如下所示。

```
void f(int i,std::string const& s);

void oops(int some_param)
{
    char buffer[1024];                        <-❶
    sprintf(buffer, "%i",some_param);
    std::thread t(f,3,buffer);                <-❷
    t.detach();
}
```

在这种情况下，正是局部变量 buffer❶的指针被传递给新线程❷，还有一个重要的时机，即函数 oops 会在缓冲在新线程上被转换为 std::string 之前退出，从而导致未定义的行为。解决之道是在将缓冲传递给 std::thread 的构造函数之前转换为 std::string。

```
void f(int i,std::string const& s);

void not_oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,std::string(buffer));   使用 std::string 避免悬浮指针
    t.detach();
}
```

在这种情况下，问题就出在你依赖从缓冲的指针到函数所期望的 std::string 对象的隐式转换，因为 std::thread 构造函数原样复制了所提供的值，并未转换为期望的参数类型。

也有可能得到相反的情况，对象被复制，而你想要的是引用。这可能发生在当线程正在更新一个通过引用传递来的数据结构时，例如，

```
void update_data_for_widget(widget_id w,widget_data& data);   <-❶

void oops_again(widget_id w)
{
    widget_data data;
    std::thread t(update_data_for_widget,w,data);   <-❷
    display_status();
    t.join();
    process_widget_data(data);                      <-❸
}
```

尽管 update_data_for_widget❶希望通过引用传递第二个参数，std::thread 的构造函数❷却并不知道，它无视函数所期望的类型，并且盲目地复制了所提供的值。当它调用 update_data_for_widget 时，它最后将传递 data 在内部的副本的引用而非对 data 自身的引用。于是，当线程完成时，随着所提供参数的内部副本的销毁，这些改动都将被舍弃，将会传递一个未改变的 data❸，而非正确更新的版本给 process_widget_data。对于熟悉 std::bind 的人来说，解决方案也是显而易见的，你需要用 std::ref 来包装确实需要被引用的参数。在这种情况下，如果你将对线程的调

用改为

```
std::thread t(update_data_for_widget,w,std::ref(data));
```

那么 update_data_for_widget 将被正确地传入 data 的引用，而非 data 副本（**copy**）的引用。

如果你熟悉 std::bind，那么参数传递语义就不足为奇，因为 std::thread 构造函数和 std::bind 的操作都是依据相同的机制定义的。这意味着，例如，你可以传递一个成员函数的指针作为函数，前提是提供一个合适的对象指针作为第一个参数。

```
class X
{
public:
    void do_lengthy_work();
};

X my_x;
std::thread t(&X::do_lengthy_work,&my_x);                    ❶
```

这段代码将在新线程上调用 my_x.do_lengthy_work()，因为 my_x 的地址是作为对象指针❶提供的。你也可以提供参数给这样的成员函数调用：std::thread 构造函数的第三个参数将作为成员函数的第一个参数等等。

提供参数的另一个有趣的场景是，这里的参数不能被复制但只能被移动（**moved**）：一个对象内保存的数据被转移到另一个对象，使原来的对象变成“空壳”。这种类型的一个例子是 std::unique_ptr，它提供了动态分配对象的自动内存管理。只有一个 std::unique_ptr 实例可以在某一时刻指向一个给定的对象，当该实例被销毁时，其指向的对象将被删除。**移动构造函数**（**move constructor**）和**移动赋值运算符**（**move assignment operator**）允许一个对象的所有权在 std::unique_ptr 实例之间进行转移（参见附录 A 中 A.1.1 节，关于移动语义的详情）。这种转移给源对象留下一个 NULL 指针。这种值的移动使得该类型的对象作为函数的参数被接受或从函数返回值。在源对象是临时的场合，这种移动是自动的，但在源是一个命名值的地方，此转移必须直接通过调用 std::move()来请求。下面的示例展示了运用 std::move 将动态对象的所有权转移到一个线程中。

```
void process_big_object(std::unique_ptr<big_object>);

std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object,std::move(p));
```

通过在 std::thread 构造函数中指定 std::move(p)，big_object 的所有权先被转移进新创建的线程的内部存储中，然后进入 process_big_object。

标准线程库中的一些类表现出与 std::unique_ptr 相同的所有权语义，std::thread 就是其中之一。虽然 std::thread 实例并不拥有与 std::unique_ptr

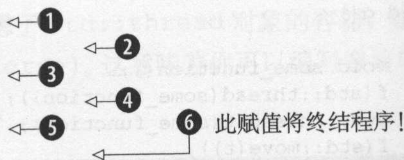
同样方式的动态对象，但他们却拥有资源，每一个实例负责管理一个执行线程。这种所有权可以在实例之间进行转移，因为 std::thread 的实例是可移动的（**movable**），即使他们不是可复制的（**copyable**）。这确保了在允许程序员选择在对象之间转换所有权的时候，在任意时刻只有一个对象与某个特定的执行线程相关联。

## 2.3 转移线程的所有权

假设你想要编写一个函数，它创建一个在后台运行的线程，但是向调用函数回传新线程的所有权，而非等待其完成，又或者你想要反过来做，创建一个线程，并将所有权传递给要等待它完成的函数。在任意一种情况下，你都需要将所有权从一个地方转移到另一个地方。

这里就是 std::thread 支持移动的由来。正如在上一节所描述的，在 C++标准库里许多拥有资源的类型，如 std::ifstream 和 std ::unique_ptr 是可移动的（**movable**），而非可复制的（**copyable**），并且 std::thread 就是其中之一。这意味着一个特定执行线程的所有权可以在 std::thread 实例之间移动，如同接下来的例子。该示例展示了创建两个执行线程，以及在三个 std::thread 实例 t1、t2 和 t3 之间对那些线程的所有权进行转移。

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
std::thread t3;
t3=std::move(t2);
t1=std::move(t3);
```




首先，启动一个新线程❶并与 t1 相关联。然后当 t2 构建完成时所有权被转移给 t2，通过调用 std::move() 来显式地转移所有权 ❷。此刻，t1 不再拥有相关联的执行线程，运行 some_function 的线程现在与 t2 相关联。

然后，启动一个新的线程并与一个临时的 std::thread 对象相关联❸。接下来将所有权转移到 t1 中，是不需要调用 std::move() 来显式移动所有权的，因为此处所有者是一个临时对象——从临时对象中进行移动是自动和隐式的。

t3 是默认构造的❹，这意味着它的创建没有任何相关联的执行线程。当前与 t2 相关联的线程的所有权转移到 t3❺，再次通过显式调用 std::move()，因为 t2 是一个命名对象。在所有这些移动之后，t1 与运行 some_other_function 的线程相关联，t2 没有相关联的线程，t3 与运行 some_function 的线程相关联。

最后一次移动❻将运行 some_function 的线程的所有权转回给 t1。但是在这种

情况下 t1 已经有了一个相关联的线程（运行着 some_other_function），所以会调用 std::terminate()来终止程序。这样做是为了与 std::thread 的析构函数保持一致。你在第 2.1.1 节曾看到，你必须在析构前显式地等待线程完成或是分离，这同样适用于赋值：你不能仅仅通过向管理一个线程的 std::thread 对象赋值一个新的值来“舍弃”一个线程。

std::thread 支持移动意味着所有权可以很容易地从一个函数中被转移出，如清单 2.5 所示。

**清单 2.5 从函数中返回 std::thread**

```
std::thread f()
{
    void some_function();
    return std::thread(some_function);
}
std::thread g()
{
    void some_other_function(int);
    std::thread t(some_other_function,42);
    return t;
}
```

同样地，如果要把所有权转移到函数中，它只能以值的形式接受 std::thread 的实例作为其中一个参数，如下所示。

```
void f(std::thread t);
void g()
{
    void some_function();
    f(std::thread(some_function));
    std::thread t(some_function);
    f(std::move(t));
}
```

std::thread 支持移动的好处之一，就是你可以建立在清单 2.3 中 thread_guard 类的基础上，同时使它实际上获得线程的所有权。这可以避免 thread_guard 对象在引用它的线程结束后继续存在所造成的不良影响，同时也意味着一旦所有权转移到了该对象，那么其他对象都不可以结合或分离该线程。因为这主要是为了确保在退出一个作用域之前线程都已完成，我把这个类称为 scoped_thread。其实现如清单 2.6 所示，同时附带一个简单的示例。

**清单 2.6 scoped_thread 和示例用法**

```
class scoped_thread
{
    std::thread t;
public:
```

```
    explicit scoped_thread(std::thread t_):    <-❶
        t(std::move(t_))
    {
        if(!t.joinable())                      <-❷
            throw std::logic_error("No thread");
    }
    ~scoped_thread()
    {
        t.join();    <-❸
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};

struct func;        <-┐ 参见清
                      │ 单2.1
void f()
{
    int some_local_state;
    scoped_thread t(std::thread(func(some_local_state)));    <-❹
    do_something_in_current_thread();
}                   <-❺
```

这个例子与清单 2.3 类似，但是新线程被直接传递到 scoped_thread❹，而不是为它创建一个单独的命名变量。当初始线程到达 f❺的结尾时，scoped_thread 对象被销毁，然后结合❸提供给构造函数❶的线程。使用清单 2.3 中的 thread_guard 类，析构函数必须检查线程是不是仍然可结合，你可以在构造函数中❷来做，如果不是则引发异常。

std::thread 对移动的支持同样考虑了 std::thread 对象的容器，如果那些容器是移动感知的（如更新后的 std::vector<>）。这意味着你可以编写像清单 2.7 中的代码，生成一批线程，然后等待它们完成。

**清单 2.7　生成一批线程并等待它们完成**

```
void do_work(unsigned id);
void f()
{
    std::vector<std::thread> threads;
    for(unsigned i=0;i<20;++i)                      ┐ 生成线程
    {                                               │
        threads.push_back(std::thread(do_work,i));  <┘
    }
    std::for_each(threads.begin(),threads.end(),    ┐ 轮流在每个线程上
                  std::mem_fn(&std::thread::join)); <┘ 调用 join()
}
```

如果线程是被用来细分某种算法的工作，这往往正是所需的。在返回调用者之前，所有线程必须全部完成。当然，清单 2.7 的简单结构意味着由线程所做的工作是自包含

的，同时它们操作的结果纯粹是共享数据的副作用。如果 f() 向调用者返回一个依赖于这些线程的操作结果的值，那么正如所写的这样，该返回值就得通过检查线程终止后的共享数据来决定。在线程间转移操作结果的替代方案将在第 4 章中讨论。

将 std::thread 对象放到 std::vector 中是线程迈向自动管理的一步。与其为那些线程创建独立的变量并直接与之结合，不如将它们视为群组。你可以进一步创建在运行时确定的动态数量的线程，更进一步地利用这一步，而不是如清单 2.7 中的那样创建固定的数量。

## 2.4 在运行时选择线程数量

C++标准库中对此有所帮助的特性是 std::thread::hardware_currency()。这个函数返回一个对于给定程序执行时能够真正并发运行的线程数量的指示。例如，在多核系统上它可能是 CPU 核心的数量。它仅仅是一个提示，如果该信息不可用则函数可能会返回 0，但它对于在线程间分割任务是一个有用的指南。

清单 2.8 展示了 std::accumulate 的一个简单的并行版本实现。它在线程之间划分所做的工作，使得每个线程具有最小数目的元素以避免过多线程的开销。请注意，该实现假定所有的操作都不引发异常，即便异常可能会发生。例如，std::thread 构造函数如果不能启动一个新的执行线程那么它将引发异常。在这样的算法中处理异常超出了这个简单示例的范围，将放在第 8 章中阐述。

**清单 2.8 std::accumulate 的简单的并行版本**

```
template<typename Iterator,typename T>
struct accumulate_block
{
    void operator()(Iterator first,Iterator last,T& result)
    {
        result=std::accumulate(first,last,result);
    }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);

    if(!length)                                            ← ❶
        return init;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;       ← ❷

    unsigned long const hardware_threads=
```

```
        std::thread::hardware_concurrency();
                                              ❸
    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    unsigned long const block_size=length/num_threads;          ❹

    std::vector<T> results(num_threads);
    std::vector<std::thread>  threads(num_threads-1);           ❺

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);                     ❻
        threads[i]=std::thread(                                 ❼
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
        block_start=block_end;                                  ❽
    }
    accumulate_block<Iterator,T>()(
        block_start,last,results[num_threads-1]);               ❾
    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));                       ❿

    return std::accumulate(results.begin(),results.end(),init); ⓫
}
```

虽然这是一个相当长的函数，但它实际上是很直观的。如果输入范围为空❶，只返回初始值 init。否则，此范围内至少有一个元素，于是你将要处理的元素数量除以最小的块大小，以获取线程的最大数量❷。这是为了避免当范围中只有五个值时，在一个32核的机器上创建32个线程。

要运行的线程数是你计算出的最大值和硬件线程数量❸的较小值。你不会想要运行比硬件所能支持的更多的线程（**超额订阅，oversubscription**），因为上下文切换将意味着更多的线程会降低性能。如果对 std::thread::hardware_concurrency() 的调用返回0，你只需简单地替换上你所选择的数量，在这个例子中我选择了2。你不会想要运行过多的线程，因为在单核的机器上这会使事情变慢，但同样地你也不希望运行的过少，因为那样的话，你就会错过可用的并发。

每个待处理的线程的条目数量是范围的长度除以线程的数量❹。如果你担心数量不能整除，没必要——稍后再来处理。

既然你知道有多少个线程，你可以为中间结果创建一个 std::vector<T>，同时为线程创建一个 std::vector<std::thread>❺。请注意，你需要启动比 num_threads 少一个的线程，因为已经有一个了。

启动线程是个简单的循环：递进 block_end 迭代器到当前块的结尾❻，并启动一个新的线程来累计此块的结果❼。下一个块的开始是这一个的结束❽。

当你启动了所有的线程后，这个线程就可以处理最后的块❾。这就是你处理所

有未被整除的地方。你知道最后一块的结尾只能是 last，无论在那个块里有多少元素。一旦累计出最后一个块的结果，你可以等待所有使用 std::for_each 生成的线程❿，如清单 2.7 中所示，接着通过最后调用 std::accumulate 将结果累加起来⓫。

在你离开这个例子前，值得指出的是在类型 T 的加法运算符不满足结合律的地方（如 float 和 double），这个 parallel_accumulate 的结果可能会跟 std::accumulate 的有所出入，这是将范围分组成块导致的。此外，对迭代器的需求要更严格一些，它们必须至少是前向迭代器（**forward iterators**），然而 std::accumulate 可以和单通输入迭代器（**input iterators**）一起工作，同时 T 必须是可默认构造的（**default constructible**）以使得你能够创建 results 向量。这些需求的各种变化是并行算法很常见的，就其本质而言，它们以某种方式的不同是为了使其并行，并且在结果和需求上产生影响。并行算法会在第 8 章中进行更深入的阐述。另外值得一提的是，因为你不能直接从一个线程中返回值，所以你必须将相关项的引用传入 results 向量中。从线程中返回结果的替代方法，会在第 4 章中通过使用 **future** 来实现。

在这种情况下，每个线程所需的所有信息在线程开始时传入，包括存储其计算结果的位置。实际情况并非总是如此。有时，作为进程的一部分有必要能够以某种方式标识线程。你可以传入一个标识数，如同在清单 2.7 中 i 的值，但是如果需要此标识符的函数在调用栈中深达数个层次，并且可能从任意线程中被调用，那样做就很不方便。当我们设计 C++线程库时就预见到了这方面的需求，所以每个线程都有一个唯一的标识符。

## 2.5 标识线程

线程标识符是 std::thread::id 类型的，并且有两种获取方式。其一，线程的标识符可以通过从与之相关联的 std::thread 对象中通过调用 get_id() 成员函数来获得。如果 std::thread 对象没有相关联的执行线程，对 get_id() 的调用返回一个默认构造的 std::thread::id 对象，表示“没有线程”。另外，当前线程的标识符，可以通过调用 std::this_thread::get_id() 获得，这也是定义在<thread>头文件中的。

std::thread::id 类型的对象可以自由地复制和比较；否则，它们作为标识符就没什么大用处。如果两个 std::thread::id 类型的对象相等，则它们代表着同一个线程，或两者都具有“没有线程”的值。如果两个对象不相等，则它们代表着不同的线程，或其中一个代表着线程，而另一个具有“没有线程”的值。

线程库不限制你检查线程的标识符是否相同，std::thread::id 类型的对象提供了一套完整的比较运算符，提供了所有不同值的总排序。这就允许它们在关系型容器中

被用作主键，或是被排序，或者任何作为程序员的你认为合适的方式进行比较。比较运算符为 std::thread::id 所有不相等的值提供了一个总的排序，所以它们表现为你直觉上期望的那样：如果 a<b 且 b<c，那么 a<c，等等。标准库还提供了 std::hash<std::thread::id>，使得 std::thread::id 类型的值可在新的无序关系型容器中作为主键来用。

std::thread::id 的实例常被用来检查一个线程是否需要执行某些操作。例如，如果线程像在清单 2.8 中那样的被用来分配工作，启动了其他线程的初始线程在需要做的工作可能会在算法中略有不同。在这种情况下，它可以在启动其他线程之前存储 std::this_thread::get_id() 的结果，然后算法的核心部分（这对所有线程都是公共的）可以对照所存储的值来检查自己的线程 ID。

```
std::thread::id master_thread;
void some_core_part_of_algorithm()
{
    if(std::this_thread::get_id()==master_thread)
    {
        do_master_thread_work();
    }
    do_common_work();
}
```

另外，当前线程的 std::thread::id 可以作为操作的一部分而存储在数据结构中。以后在相同数据结构上的操作可以对照执行此操作的线程 ID 来检查所存储的 ID，来确定哪些操作是允许的/需要的。

类似地，线程 ID 可以指定的数据需要与一个线程进行关联，并且诸如线程局部存储这样的替代机制不适用的地方，用作关系型容器的主键。例如这样的一个容器，它可以被控制线程用来存储关于在它控制下的每个线程的信息，或是在线程之间传递信息。

这种想法就是，在大多数情况下，std::thread::id 足以作为线程的通用标识符。只有当标识符具有与其相关联的语义（比如作为数组的索引）时，才有必要用替代方案。你甚至可以将一个 std::thread::id 实例写到诸如 std::cout 这样的输出流中。

```
std::cout<<std::this_thread::get_id();
```

你得到的确切的输出，严格取决于实现，标准给定的唯一保证是，比较结果相等的线程 ID 应该产生相同的输出，而那些比较结果不相等的应该给出不同的输出。因此，这主要是对调试和日志有用，但数值是没有语义的，所以也没有更多可说的了。

## 2.6 小结

在这一章中，我介绍了线程管理与 C++标准库的基本知识：启动线程，等待其完成，以及因为你希望它们在后台运行而不等待其完成。你还看到了如何在线程开始时将参数

传递给线程函数，如何将管理线程的责任从代码的一个部分转移到另一个部分，以及如何用线程组来做分配工作。最后，我讨论了标识线程，以便关联数据或是不方便通过替代方法进行关联的特定线程的行为。尽管你可以使用运行在独立数据上的完全独立的线程做很多事情，例如在清单 2.8 中那样，但有些时候，当线程运行时在它们之间共享数据是更理想的。第 3 章围绕直接在线程间共享数据进行讨论，而第 4 章围绕有或没有共享数据的同步操作涵盖更一般性的问题。

# 第3章 在线程间共享数据

本章主要内容
- 线程间共享数据的问题
- 用互斥元保护数据
- 用于保护共享数据的替代工具

将线程用作并行的关键优点之一，是在它们之间简单、直接地共享数据的潜力。所以既然我们已经介绍了启动和管理线程，现在让我们来看看共享数据的相关问题。

假设你正与朋友合租着一套公寓。公寓里只有一个厨房和一间浴室。除非你们特别的友好，否则你们不能同时使用浴室。如果你的室友长时间占据着浴室，当你需要用它时就会很不爽。同样地，尽管两人同时做饭也是可以的，但若是你们有一个组合烤箱，如果你们中的一个想要烤香肠而另一个想要烘蛋糕，这就没办法皆大欢喜。此外，大家都知道共享空间的挫败感，以及一个任务做到一半才发现有人借走了我们需要的东西，或者某件东西无意中被别人改变了。

这和线程是相同的。如果你在线程之间共享数据，你需要设置规则，哪个线程可以访问数据的哪一位，什么时间以及如何将更改传达给关心此数据的其他线程。在单个进程中的多个线程之间可以轻易地共享数据不单纯是优点——它也可以是个很大的缺点。共享数据的不正确使用是与并发有关的错误的最大诱因之一，造成的后果可比香肠味道的蛋糕糟多了。

本章涉及了在 C++线程间安全地共享数据，避免可能出现的潜在问题，同时使

收益最大化。

## 3.1　线程之间共享数据的问题

从整体上来看，所有线程之间共享数据的问题，都是修改数据导致的。如果所有的共享数据都是只读的，就没有问题，因为一个线程所读取的数据不受另一个线程是否正在读取相同的数据而影响（**If all shared data is read-only, there's no problem,because the data read by one thread is unaffected by whether or not another thread is reading the same data**）。然而，如果数据是在线程之间共享的，同时一个或多个线程开始修改数据，就可能有很多的麻烦。在这种情况下，你必须要小心确保一切安好。

一个被广泛用来帮助程序员推导代码的概念，就是不变量（**invariants**）——对于特定的数据结构总是为真的语句，例如“此变量包含了列表中项目的数量。”这些不变量在更新中经常被打破，尤其是在数据结构比较复杂或是更新需要修改超过一个值时。

考虑一个双向链表，它的每一个节点持有指向表中下一节点和前一节点的指针。其中一个不变量就是如果你跟随从一个节点（A）到另一个节点（B）的“下一个”指针，则那个节点（B）的“前一个”指针指回到前一个节点（A）。为了从表中删除一个节点，两边的节点都必须被更新为指向彼此。一旦其中一个被更新，直到另一侧的节点也被更新前不变量是打破的，当更新完成后，再次持有不变量。

从这样的表中删去一个条目的步骤如图 3.1 所示。

**1**　标识要删除的节点（N）。

**2**　将 N 的前一节点到 N 的链接更新为指向 N 的后一节点。

**3**　将 N 的后一节点到 N 的链接更新为指向 N 的前一节点。

**4**　删除节点 N。

如你所见，在步骤 b 和 c 之间，在一个方向上的链接与在相反的方向上的链接不一致，并且不变量损坏。

修改线程之间共享数据的最简单的潜在问题就是破坏不变量。如果你没有为确保其他情况而做些特别的工作，要是一个线程正在读取双向链表，而另一个线程正在删除一个节点，那么读线程很有可能看到一个节点仅被部分删除了的链表（因为在图 3.1 的步骤 b 中，只有其中一个链接被改变了），因此不变量损坏。不变量损坏的后果可能有所不同，如果其他线程只是在图中由左到右读取链表项，它会跳过正被删除的节点。另一方面，如果又一个线程试图删除图中最右边的节点，则可能最终永久性破坏数据结构，并使得程序崩溃。无论结果如何，这是并发代码中错误的最常见诱因之一的例子：竞争条件（**race condition**）。

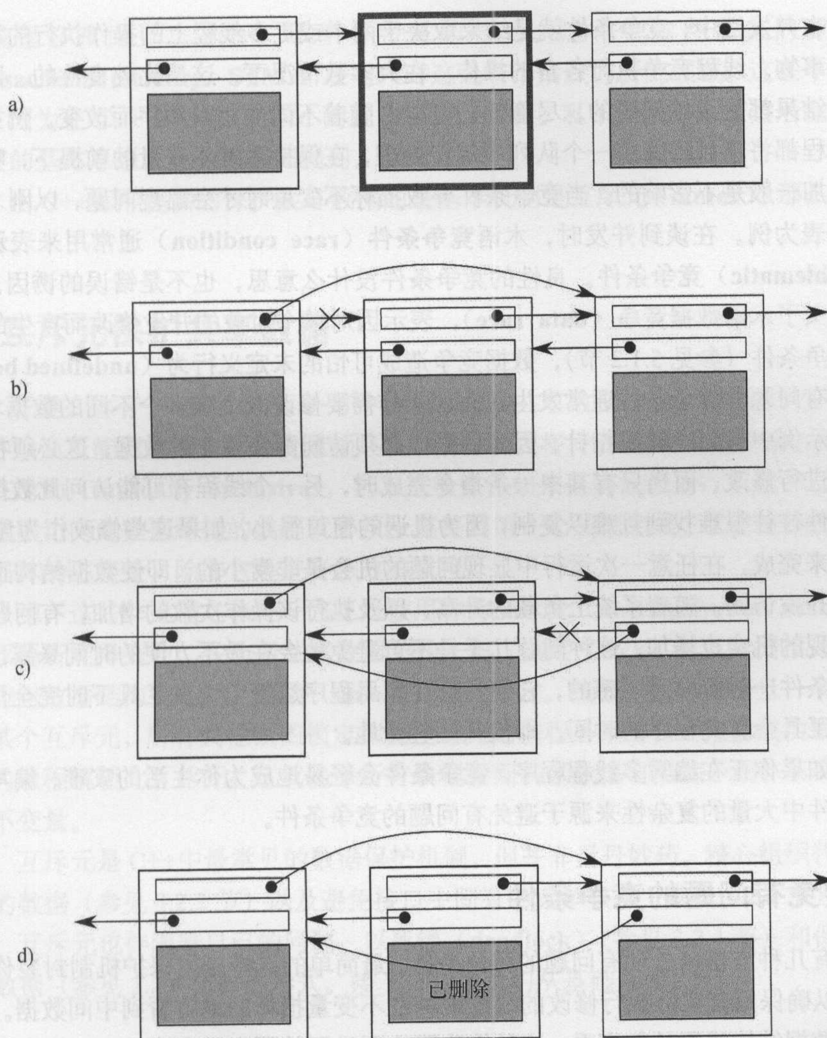



图 3.1　从双向链表中删除一个节点

## 3.1.1　竞争条件

假设你在电影院买票看电影。如果是个大电影院，会有多个收银员收款，所以不止一个人可以同时买票。如果有人在另一个收银台也购买了与你同一部电影的票，这时可供你选择的座位取决于事实上是其他人先订购还是你先订购。如果只剩少量座位，这种差异可能会很关键。字面上可以看作一个比赛，看谁得到最后的电影票。这是一个竞争条件（**race condition**）的例子：得到哪个座位（或者甚至是否得到票）取决于两次购买的相对顺序。

在并发性中，竞争条件就是结果取决于两个或更多线程上的操作执行的相对顺序的一切事物。线程竞争执行各自的操作。在大多数情况下，这是比较良性的，因为所有可能的结果都是可以接受的，尽管他们可能会随着不同的相对顺序而改变。例如，如果两个线程都将项目添加到一个队列中等待处理，在保持系统不变量的前提下，哪个项目先被添加一般是不影响的。当竞争条件导致损坏不变量时才会出现问题，以刚才提到的双向链表为例。在谈到并发时，术语**竞争条件**（**race condition**）通常用来表示**有问题的**（**problematic**）竞争条件。良性的竞争条件没什么意思，也不是错误的诱因。C++标准还定义了术语**数据竞争**（**data race**），表示因对单个对象的并发修改而产生的特定类型的竞争条件（参见 5.1.2 节），数据竞争造成可怕的**未定义行为**（**undefined behavior**）。

有问题的竞争条件通常发生在完成操作需要修改两个或多个不同的数据块的地方，就如示例中的两个链表指针。因为该操作必须访问两块独立的数据，这必须在单独的指令中进行修改，而当只有其中一条指令完成时，另一个线程有可能访问此数据结构。竞争条件往往很难找到且难以复制，因为机遇的窗口很小。如果这些修改作为连续的 CPU 指令来完成，在任意一次运行中显现问题的机会是非常小的，即使数据结构正被另一个线程并发访问。随着系统上负载的升高，以及执行该操作次数的增加，有问题的执行序列出现的机会也增加。这种问题几乎是不可避免地会在最不方便的时间暴露出来。由于竞争条件一般是时间敏感的，它们常常在应用程序运行于调试工具下时完全消失，因为调试工具会影响程序的时间，即使只是轻微地。

如果你正在编写多线程程序，竞争条件会轻易地成为你生活的灾难。编写使用并发的软件中大量的复杂性来源于避免有问题的竞争条件。

### 3.1.2　避免有问题的竞争条件

有几种方法来处理有问题的竞争条件。最简单的选择是用保护机制封装你的数据结构，以确保只有实际执行修改的线程能够在不变量损坏的地方看到中间数据。从其他访问该数据结构线程的角度看，这种修改要么还没开始要么已完成。C ++标准库提供了一些这样的机制，在本章中均有述及。

另一个选择是修改数据结构的设计及其不变量，从而令修改作为一系列不可分割的变更来完成，每个修改均保留其不变量。这通常被称为**无锁编程**（**lock-free programming**），且难以尽善尽美。如果你工作在这个级别上，内存模型的细微差异和确认哪些线程可能看到哪组值，会变得很复杂。内存模型在第 5 章中阐述，而无锁编程在第 7 章中进行讨论。

处理竞争条件的另一种方式是将对数据结构的更新作为一个**事务**（**transaction**）来处理，就如同在一个事务内完成数据库的更新一样。所需的一系列数据修改和读取被存储在一个事务日志中，然后在单个步骤中进行提交。如果该提交因为数据结构已被另一

个线程修改而无法进行，该事务将重新启动。这称为**软件事务内存**（**software transactional memory**，**STM**），在写作时这是一个活跃的研究领域。这在本书中不会述及，因为在C++中没有对STM的直接支持。然而，私下里做些事情然后在单个步骤中提交的基本思想，我会在后面提到。

由C++标准提供的保护共享数据的最基本机制是**互斥元**（**mutex**），那么我们先来看一看。

## 3.2 用互斥元保护共享数据

于是，你有一个类似于上一节中链表那样的共享数据结构，你想要保护它免于竞争条件以及可能因此产生的不变量损坏。如果你可以将所有访问该数据结构的代码块标记为互斥的（**mutually exclusive**），岂不是很好？如果任何线程运行了其中之一，所有其他试图访问此数据结构的线程就必须一直等到第一个线程完成。这就使得线程不可能看到损坏的不变量，除非它是进行修改的线程。

嗯，这并非无稽之谈——它正是使用称为**互斥元**（**mutex, mutual exclusion**）的同步原语所能得到的。在访问共享数据结构之前，**锁定**（**lock**）与该数据相关的互斥元，当访问数据结构完成后，**解锁**（**unlock**）该互斥元。线程库会确保一旦一个线程已经锁定某个互斥元，所有其他试图锁定相同互斥元的线程必须等待，直到成功锁定了该互斥元的线程解锁此互斥元。这就确保所有线程看到共享数据自洽的一面，不带有任何损坏的不变量。

互斥元是C++中最常见的数据保护机制，但并非灵丹妙药。精心组织代码来保护正确的数据（参见3.2.2节）以及避免接口中固有的竞争条件（参见3.2.3节）也是很重要的。互斥元也伴随着自己的问题，以**死锁**（**deadlock**）（参见3.2.4节）和保护过多或过少数据（参见3.2.8节）的形式。接下来，让我们从基础开始。

### 3.2.1 使用C++中的互斥元

在C++中，通过构造`std::mutex`的实例创建互斥元，调用成员函数`lock()`来锁定它，调用成员函数`unlock()`来解锁它。然而，直接调用成员函数是不推荐的做法，因为这意味着你必须记住在离开函数的每条代码路径上都调用`unlock()`，包括由于异常所导致的在内。作为替代，标准C ++库提供了`std::lock_guard`类模板，实现了互斥元的RAII惯用语法；它在构造时锁定所给的互斥元，在析构时将互斥元解锁，从而保证被锁定的互斥元始终被正确解锁。清单3.1的代码展示了如何使用`std::mutex`保护一个可被多个线程访问的列表，连同`std::lock_guard`。二者都声明于`<mutex>`头文件中。

清单 3.1 用互斥元保护列表

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list;      <-❶
std::mutex some_mutex;         <-❷

void add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex);    <-❸
    some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{
    std::lock_guard<std::mutex> guard(some_mutex);    <-❹
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

在清单 3.1 中，有一个全局变量❶，它被相应的 std::mutex 的全局实例❷保护。在 add_to_list()❸以及list_contains()❹中对std::lock_guard<std::mutex>的使用意味着这些函数中的访问是互斥的，list_contains()将无法在 add_to_list()进行修改的半途中看到该列表。

尽管这种全局变量的使用偶尔也是恰当的，在大多数情况下，不用全局变量，而是在类中将互斥元和受保护的数据组织在一起，是很普遍的。这是一个标准的面向对象应用程序设计规则，通过将它们放在一个类中，清楚地标记他们是相关的，还可以封装函数以及强制保护。在这种情况下，函数 add_to_list 和 list_contains 将成为类的成员函数，互斥元和受保护的数据都作为类的 private 成员，使其更容易鉴别哪些代码可以访问数据，哪些代码需要锁定互斥元。如果类的所有成员函数在访问任意其他数据成员之前锁定互斥元，并且在操作完成时解锁，则数据对于所有的访问者都被很好地保护了。

其实，并不完全是那样，你将敏锐地发现，如果其中一个成员函数返回对受保护数据的指针或引用，那么所有成员函数都以良好顺序的方式锁定互斥元也是没关系的，因为你已在保护中捅了一个大窟窿。**能够访问（并可能修改）该指针或引用的任意代码现在可以访问受保护的数据而无需锁定该互斥元。**因此使用互斥元保护数据需要仔细设计接口，以确保在有任意对受保护的数据进行访问之前，互斥元已被锁定，且不留后门。

### 3.2.2 为保护共享数据精心组织代码

如你所见，用互斥元保护数据并不只是像在每个成员函数中拍进一个 std::lock_guard 对象那样容易，一个迷路的指针或引用，所有的保护都将白费。在一个层面上，

检查迷路的指针或引用是容易的，只要没有一个成员函数通过其返回值或输出参数，返回受保护数据的指针或引用给其调用者，数据就安全了。如果更深入一些，它没有那么直观——远远没有。除了检查成员函数没有向其调用者传出指针和引用，检查它们没有向其调用的不在你掌控之下的函数传入这种指针和引用，也是很重要的。这同样危险，那些函数可能将指针和引用存储在某个地方，将来可以脱离互斥元的保护而被使用。在这方面特别危险的是，函数是通过函数参数或其他方式在运行时提供的，如清单 3.2 所示。

**清单 3.2　意外地传出对受保护数据的引用**

```
class some_data
{
    int a;
    std::string b;
public:
    void do_something();
};

class data_wrapper
{
private:
    some_data data;
    std::mutex m;
public:
    template<typename Function>
    void process_data(Function func)
    {
        std::lock_guard<std::mutex> l(m);
        func(data);              <-- ❶ 传递“受保护的”数据到用户提供的函数
    }
};

some_data* unprotected;

void malicious_function(some_data& protected_data)
{
    unprotected=&protected_data;
}

data_wrapper x;

void foo()
{
    x.process_data(malicious_function);    <-- ❷ 传入一个恶意函数
    unprotected->do_something();           <-- ❸ 对受保护的数据进行未受保护的访问
}
```

在这个例子中，process_data 中的代码看起来挺无害，受到 std::lock_guard 很好地保护，但对用户提供的函数 func 的调用❶意味着 foo 可以传入 malicious_function❷ 来绕过保护，然后无需锁定互斥元即可调用 do_something()❸。

从根本上说，这个代码的问题在于它没有完成你所设置的内容，标记所有访问该数据结构的代码为互斥的（**mutually exclusive**）。在这个例子中，忽略了 foo()中调用 unprotected->do_something()的代码。不幸的是，这部分问题不是 C++线程库所能帮助你的，而这取决于作为程序员的你，去锁定正确的互斥元来保护你的数据。想想好的一面，你有了一个可遵循的准则，它会在这些情况下帮助你：**不要将对受保护数据的指针和引用传递到锁的范围之外，无论是通过从函数中返回它们、将其存放在外部可见的内存中，还是作为参数传递给用户提供的函数。**

虽然这是在试图使用互斥元来保护共享数据时常犯的错误，但这绝非唯一可能的隐患。在下一节中你会看到，可能仍然会有竞争条件，即便当数据被互斥元保护着。

### 3.2.3 发现接口中固有的竞争条件

仅仅因为使用了互斥元或其他机制来保护共享数据，未必会免于竞争条件，你仍然需要确定保护了适当的数据。再次考虑双向链表的例子。为了让线程安全地删除节点，你需要确保已阻止对三个结点的并发访问。要删除的节点及其两边的结点。如果你分别保护访问每个节点的指针，就不会比未使用互斥元的代码更好，因为竞争条件仍会发生——需要保护的不是个别步骤中的个别结点，而是整个删除操作中的整个数据结构。这种情况下最简单的解决办法，就是用单个互斥元保护整个列表，如清单 3.1 中所示。

仅仅因为在列表上的个别操作是安全的，你还没有摆脱困境。你仍然会遇到竞争条件，即便是一个非常简单的接口。考虑像 std::stack 容器适配器这样的堆栈数据结构，如清单 3.3 中所示。除了构造函数和 swap()，对 std::stack 你只有五件事情可以做：可以 push()一个新元素入栈、pop()一个元素出栈、读 top()元素、检查它是否 empty()以及读取元素数量——堆栈的 size()。如果更改 top()使得它返回一个副本，而不是引用（这样你就遵循了 3.2.2 节的准则），同时用互斥元保护内部数据，该接口依然固有地受制于竞争条件。这个问题对基于互斥元的实现并不是独一无二的。它是一个接口问题，因此对于无锁实现仍然会发生竞争条件。

**清单 3.3　std::stack 容器适配器的接口**

```
template<typename T,typename Container=std::deque<T> >
class stack
{
public:
    explicit stack(const Container&);
    explicit stack(Container&& = Container());
    template <class Alloc> explicit stack(const Alloc&);
    template <class Alloc> stack(const Container&, const Alloc&);
    template <class Alloc> stack(Container&&, const Alloc&);
    template <class Alloc> stack(stack&&, const Alloc&);
```

```
    bool empty() const;
    size_t size() const;
    T& top();
    T const& top() const;
    void push(T const&);
    void push(T&&);
    void pop();
    void swap(stack&&);
};
```

这里的问题是 empty()的结果和 size()不可靠。虽然它们可能在被调用时是正确的，一旦它们返回，在调用了 empty()或 size()的线程可以使用该信息之前，其他线程可以自由地访问堆栈，并且可能 push()新元素入栈或 pop()已有的元素出栈。

特别地，如果该 stack 实例是**非共享的**，如果栈非空，检查 empty()并调用 top()访问顶部元素是安全的，如下所示。

```
stack<int> s;
if(!s.empty())                  <-❶
{
    int const value=s.top();    <-❷
    s.pop();                    <-❸
    do_something(value);
}
```

它不仅在单线程代码中是安全的，预计为：在空堆栈上调用 top()是未定义的行为。对于共享的 stack 对象，**这个调用序列不再安全**，因为在调用 empty()❶和调用 top()❷之间可能有来自另一个线程的 pop()调用，删除最后一个元素。因此，这是一个典型的竞争条件，为了保护栈的内容而在内部使用互斥元，却并未能将其阻止，这就是接口的影响。

怎么解决呢？发生这个问题是接口设计的后果，所以解决办法就是改变接口。然而，这仍然回避了问题，要作出什么样的改变？在最简单的情况下，你只要声明 top()在调用时如果栈中没有元素则引发异常。虽然这直接解决了问题，但它使编程变得更麻烦，因为现在你得能捕捉异常，即使对 empty()的调用返回 false。这基本上使得 empty()的调用变得纯粹多余。

如果你仔细看看前面的代码片段，还有另一个可能的竞争条件，但这一次是在调用 top()❷ 和调用 pop()❸ 之间。考虑运行着前面代码片段的两个线程，它们都引用着同一个 stack 对象 s。这并非罕见的情形，当为了性能而使用线程时，有数个线程在不同的数据上运行相同的代码是很常见的，并且一个共享的 stack 对象非常适合用来在它们之间分隔工作。假设一开始栈里有两个元素，那么你不用担心在任一线程上的 empty()和 top()之间的竞争，只需考虑可能的执行模式。

如果栈从内部被互斥元保护，只有一个线程可以在任何时间运行栈的成员函数，那么这些调用就能得以很好地交错，而对 do_something()的调用可以同时运行。一个可能的执行正如表 3.1 所示。

表 3.1 两个线程堆栈上可能的操作顺序

| 线程 A | 线程 B |
| --- | --- |
| if(!s.empty()) | |
| | if(!s.empty()) |
| int const value = s.top(); | |
| | int const value = s.top(); |
| s.pop(); | |
| do_something(value); | s.pop(); |
| | do_something(value); |

如你所见，如果这些是仅有的在运行的线程，在两次调用 top()修改该栈之间没有任何东西，所以这两个线程将看到相同的值。不仅如此，在 pop()的两次调用之间没有对 top()的调用。因此，栈上的两个值其中一个还没被读取就被丢弃了，而另一个被处理了两次。这是另一种竞争条件，远比 empty()/top()竞争的未定义行为更糟糕。从来没有任何明显的错误发生，同时错误造成的后果可能和诱因差距甚远，尽管他们明显取决于 do_something()到底做什么。

这要求对接口进行更加激进的改变，在互斥元的保护下结合对 top()和 pop()两者的调用。Tom Cargill[1]指出，如果栈上对象的拷贝构造函数能够引发异常，结合调用可能会导致问题。从 Herb Sutter[2]的异常安全的观点来看，这个问题被处理得较为全面，但潜在的竞争条件为这一结合带来了新的东西。

对于那些尚未意识到这个问题的人，考虑一下 stack<vector<int>>。现在，vector 是一个动态大小的容器，所以当你复制 vector 时，为了复制其内容，库就必须从堆中分配更多的内存。如果系统负载过重，或有明显的资源约束，此次内存分配就可能失败，于是 vector 的拷贝构造函数可能引发 std::bad_alloc 异常。如果 vector 中含有大量的元素的话则尤其可能。如果 pop()函数被定义为返回出栈值，并且从栈中删除它，就会有潜在的问题。仅在栈被修改后，出栈值才返回给调用者，但复制数据以返回给调用者的过程可能会引发异常。如果发生这种情况，刚从栈中出栈的数据会丢失，它已经从栈中被删除了，但该复制却没成功！std::stack 接口的设计者笼统地将操作一分为二。获取顶部的元素（top()），然后将其从栈中删除（pop()），以致于你无法安全地复制数据，它将留在栈上。如果问题是堆内存不足，也许应用程序可以释放一些内存，然后再试一次。

不幸的是，这种划分正是你在消除竞争条件中试图去避免的！值得庆幸的是，还有替代方案，但他们并非无代价的。

---

1 Tom Cargill, *Exception Handling: A False Sense of Security*, in C++ Report 6, no. 9 (November–December 1994). Also available at http://www.informit.com/content/images/020163371x/ supplements/ Exception_Handling_Article.html

2 Herb Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* (Addison Wesley Professional, 1999)

## 1. 选项 1：传入引用

第一个选项是把你希望接受出栈值的变量的引用，作为参数传递给对 pop() 的调用。

```
std::vector<int> result;
some_stack.pop(result);
```

这在很多情况下都适用，但它有个明显的缺点，要求调用代码在调用之前先构造一个该栈值类型的实例，以便将其作为目标传入。对于某些类型而言这是行不通的，因为构造一个实例在时间和资源方面是非常昂贵的。对于其他类型，这并不总是可能的，因为构造函数需要参数，而在代码的这个位置不一定可用。最后，它要求所存储的类型是可赋值的。这是一个重要的限制。许多用户定义的类型不支持赋值，尽管它们可能支持移动构造函数，或者甚至是拷贝构造函数（从而允许通过值来返回）。

## 2. 选项 2：要求不引发异常的拷贝构造函数或移动构造函数

对于有返回值的 pop() 而言只有一个异常安全问题，就是以值进行的返回可能引发异常。许多类型具有不引发异常的拷贝构造函数，并且在 C++标准中有了新的右值引用的支持（参见附录 A 中 A.1 节），越来越多的类型将不会引发异常的移动构造函数，即便他们的拷贝构造函数会如此。一个有效的选择，就是把对线程安全堆栈的使用，限制在能够安全地通过值来返回且不引发异常的类型之内。

虽然这样安全了，但并不理想。尽管你可以在编译时使用 std::is_nothrow_copy_constructible 和 std::is_nothrow_move_constructible 类型特征，来检测一个不引发异常的拷贝或移动构造函数的存在，但这却很受限制。相比于具有不能引发异常的拷贝和/或移动构造函数的类型，有更多的用户定义类型具有能够引发异常的拷贝构造函数且没有移动构造函数（尽管这会随着人们习惯了 C++11 中对右值引用的支持而改变）。如果这种类型不能被存储在你的线程安全堆栈中，是不幸的。

## 3. 选项 3：返回指向出栈项的指针

第三个选择是返回一个指向出栈项的指针，而非通过值来返回该项。其优点是指针可以被自由地复制而不会引发异常，这样你就避免了 Cargill 的异常问题。其缺点是，返回一个指针时需要一种手段来管理分配给对象的内存，对于像整数这样简单的类型，这种内存管理的成本可能会超过仅通过值来返回该类型。对于任何使用此选项的接口，std::shared_ptr 会是指针类型的一个好的选择。它不仅避免了内存泄漏，因为一旦最后一个指针被销毁则该对象也会被销毁，并且库可以完全控制内存分配方案且不必使用 new 和 delete。对于优化用途来说这是很重要的，要求用 new 分别分配堆栈中的每一个对象，会比原来非线程安全的版本带来大得多的开销。

### 4. 选项 4：同时提供选项 1 以及 2 或 3

灵活性永远不应被排除在外，特别是在通用的代码中。如果你选择选项 2 或 3，那么同时提供选项 1 也是相对容易的，这也为你的代码的用户提供了选择的能力，为了很小的额外成本，哪个选项对他们是最适合的。

### 5. 一个线程安全堆栈的示范定义

清单 3.4 展示了在接口中没有竞争条件的栈的类定义，实现了选项 1 和 3。pop() 有两个重载，一个接受存储该值的位置的引用，另一个返回 std:: shared_ptr<>。它具有一个简单的接口，只有两个函数，push() 和 pop()。

**清单 3.4 一个线程安全栈的概要类定义**

```
#include <exception>
#include <memory>                           <-- For std::shared_ptr<>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class threadsafe_stack                                   赋值运算符被删除了 ❶
{
public:
    threadsafe_stack();
    threadsafe_stack(const threadsafe_stack&);
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;
    void push(T new_value);
    std::shared_ptr<T> pop();
    void pop(T& value);
    bool empty() const;
};
```

通过削减接口，你考虑到了最大的安全性，甚至对整个堆栈的操作都受到限制。栈本身不能被赋值，因为赋值运算符被删除❶（参见附录 A 中 A.2 节），而且也没有 swap() 函数。然而，它可以被复制，假设栈的元素可以被复制。如果栈是空的，pop() 函数引发一个 empty_stack 异常，所以即使在调用 empty() 后栈被修改，一切仍将正常工作。正如选项 3 的描述中提到的，如果需要，std::shared_ptr 的使用允许栈来处理内存分配问题，同时避免对 new 和 delete 的过多调用。五个堆栈操作现在变成三个，push()、pop() 和 empty()。甚至 empty() 都是多余的。接口的简化可以更好地控制数据。你可以确保互斥元为了操作的整体而被锁定。清单 3.5 展现了一个简单的实现，一个围绕 std::stack<>的封装器。

**清单 3.5 一个线程安全栈的详细类定义**

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;                              // ❶ 在构造函数体中执行复制
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(new_value);
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();         // 在试着出栈值的时候检查是否为空
        std::shared_ptr<T> const res(std::make_shared<T>(data.top()));  // 在修改栈之前分配返回值
        data.pop();
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=data.top();
        data.pop();
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

这个栈的实现实际上是可复制的（**copyable**）——源对象中的拷贝构造函数锁定互

斥元，然后复制内部栈。你在构造函数体中进行复制❶而不是成员初始化列表，以确保互斥元被整个副本持有。

top()和 pop()的讨论表明，接口中有问题的竞争条件基本上因为锁定的粒度过小而引起。保护没有覆盖期望操作的整体。互斥元的问题也可以由锁定的粒度过大而引起；极端情况是单个的全局互斥元保护所有共享的数据。在一个有大量共享数据的系统中，这可能会消除并发的所有性能优势，因为线程被限制为每次只能运行一个，即便是在他们访问数据的不同部分的时候。被设计为处理多处理器系统的 Linux 内核的第一个版本，使用了单个全局内核锁。虽然这也能工作，但却意味着一个双处理器系统通常比两个单处理器系统的性能更差，四个处理器系统的性能远远没有四个单处理器系统的性能好。有太多对内核的竞争，因此在更多处理器上运行的线程无法进行有效的工作。Linux 内核的后续版本已经转移到一个更细粒度的锁定方案，因而四个处理器的系统性能更接近理想的单处理器系统的 4 倍，因为竞争少得多。

细粒度锁定方案的一个问题，就是有时为了保护操作中的所有数据，需要不止一个互斥元。如前所述，有时要做的正确的事情是增加被互斥元所覆盖的数据粒度，以使得只需要一个互斥元被锁定。然而，这有时是不可取的，例如互斥元保护着一个类的各个实例。在这种情况下，在下个级别进行锁定，将意味着要么将锁丢给用户，要么就让单个互斥元保护该类的所有实例，这些都不甚理想。

如果对于一个给定的操作你最终需要锁定两个或更多的互斥元，还有另一个潜在的问题潜伏在侧：**死锁（deadlock）**。这几乎是竞争条件的反面，两个线程不是在竞争成为第一，而是每一个都在等待另外一个，因而都不会有任何进展。

### 3.2.4　死锁：问题和解决方案

试想一下，你有一个由两部分组成的玩具，并且你需要两个部分一起玩——例如，玩具鼓和鼓槌。现在，假设你有两个小孩，他们两人都喜欢玩它。如果其中一人同时得到鼓和鼓槌，那这个孩子就可以高兴地玩鼓，直到厌烦。如果另一个孩子想要玩，就得等，不管这让他多不爽。现在想象一下，鼓和鼓槌被（分别）埋在玩具箱里，你的孩子同时都决定玩它们，于是他们去翻玩具箱。其中一个发现了鼓，而另一个发现了鼓槌。现在他们被困住了，除非一人让另一人玩，不然每个人都会赖着他已有的东西，并要求另一人将另一部分给自己，否则就都玩不成。

现在想象一下，你没有抢玩具的孩子，但却有争夺互斥元的线程。一对线程中的每一个都需要同时锁定两个互斥元来执行一些操作，并且每个线程都拥有了一个互斥元，同时等待另外一个。线程都无法继续，因为每个线程都在等待另一个释放其互斥元。这种情景称为**死锁（deadlock）**，它是在需要锁定两个或更多互斥元以执行操作时的最大问题。

为了避免死锁，常见的建议是始终使用相同的顺序锁定这两个互斥元。如果你总是在互斥元B之前锁定互斥元A，那么你永远不会死锁。有时候这是很直观的，因为互斥元服务于不同的目的，但其他时候却并不那么简单，比如当互斥元分别保护相同类的各个实例时。例如，考虑同一个类的两个实例之间的数据交换操作，为了确保数据被正确地交换，而不受并发修改的影响，两个实例上的互斥元都必须被锁定。然而，如果选择了一个固定的顺序（例如，作为第一个参数提供的实例的互斥元，然后是作为第二个参数所提供的实力的互斥元），可能适得其反：它表示两个线程尝试通过交换参数，而在相同的两个实例之间交换数据，你将产生死锁。

幸运的是，C++标准库中的 std::lock 可以解决这一问题——std::lock 函数可以同时锁定两个或更多的互斥元，而没有死锁的风险。清单 3.6 中的例子展示了如何使用它来完成简单的交换操作。

**清单 3.6 在交换操作中使用 std::lock()和 std::lock_guard**

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);

class X
{
private:
    some_big_object some_detail;
    std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd){}

    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
            return;
        std::lock(lhs.m,rhs.m);                                    ❶
        std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);  ❷
        std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);  ❸
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

首先，检查参数以确保它们是不同的实例，因为试图在你已经锁定了的 std::mutex 上获取锁，是未定义的行为。(允许同一线程多重锁定的互斥元类型为 std::recursive_mutex。详见 3.3.3 节）然后，调用 std::lock()锁定这两个互斥元❶，同时构造两个 std::lock_guard 的实例 ❷❸，每个实例对应一个互斥元。额外提供一个参数 std::adopt_lock 给互斥元，告知 std::lock_guard 对象该互斥元已被锁定，并且它们只应沿用互斥元上已有锁的所有权，而不是试图在构造函数中锁定互斥元。

这就确保了通常在受保护的操作可能引发异常的情况下，函数退出时正确地解锁互斥元，这也考虑到了简单返回。此外，值得一提的是，在对 std::lock 的调用中锁定

lhs.m 抑或是 rhs.m 都可能引发异常，在这种情况下，该异常被传播出 std::lock。如果 std::lock 已经成功地在一个互斥元上获取了锁，当它试图在另一个互斥元上获取锁的时候，就会引发异常，前一个互斥元将会自动释放。std::lock 提供了关于锁定给定的互斥元的全或无的语义。

尽管 std::lock 能够帮助你在需要同时获得两个或更多锁的情况下避免死锁，但是如果要分别获取锁，就没用了。在这种情况下，你必须依靠你作为开发人员的戒律，以确保不会得到死锁。这谈何容易，死锁是在编写多线程代码时遇到的最令人头疼的问题之一，而且往往无法预测，大部分时间内一切都工作正常。然而，有一些相对简单的规则可以帮助你写出无死锁的代码。

### 3.2.5 避免死锁的进一步指南

死锁并不仅仅产生于锁定，虽然这是最常见的诱因。你可以通过两个线程来制造死锁，不用锁定，只需令每个线程在 std::thread 对象上为另一线程调用 join()。在这种情况下，两个线程都无法取得进展，因为正等着另一个线程完成，就像孩子们争夺他们的玩具。这种简单的循环可以发生在任何地方，一个线程等待另一个线程执行一些动作而另一个线程同时又在等待第一个线程，而且这不仅限于两个线程，三个或更多线程的循环也会导致死锁。避免死锁的准则全都可以归结为一个思路，如果有另外一个线程有可能在等待你，那你就别等它。这个独特的准则为识别和消除别的线程等待你的可能性提供了方法。

#### 1. 避免嵌套锁

第一个思路是最简单的，如果你已经持有一个锁，就别再获取锁。如果你坚持这个准则，光凭使用锁是不可能导致死锁的，因为每个线程仅持有一个锁。你仍然会从其他事情（像是线程相互等待）中得到死锁，但是互斥元锁定可能死锁最常见的诱因。如果需要获取多个锁，为了避免死锁，就以 std::lock 的单个动作来实行。

#### 2. 在持有锁时，避免调用用户提供的代码

这是前面一条准则的简单后续。因为代码是用户提供的，你不知道它会做什么，它可能做包括获取锁在内的任何事情。如果你在持有锁时调用用户提供的代码，并且这段代码获取一个锁，你就违反了避免嵌套锁的准则，可能导致死锁。有时候这是无法避免的。如果你在编写泛型代码，如 3.2.3 节中的堆栈，在参数类型上的每一个操作都是用户提供的代码。在这种情况下，你需要新的准则。

#### 3. 以固定顺序获取锁

如果你绝对需要获取两个或更多的锁，并且不能以 std::lock 的单个操作取得，

次优的做法是在每个线程中以相同的顺序获取它们。我在 3.2.4 节中曾谈及此点，是作为在获取两个互斥元时避免死锁的方法，关键是要以一种在线程间相一致的方法来定义其顺序。在某些情况下，这是相对简单的。例如，看一看 3.2.3 节中的堆栈——互斥元在每个栈实例的内部，但对于存储在栈中的数据项的操作，则需要调用用户提供的代码。然而，你可以添加约束，对于存储在栈中的数据项的操作，都不应对栈本身进行任何操作。这样就增加了栈的使用者的负担，但是将数据存储在一个容器中来访问该容器是很罕见的，并且一旦发生就会十分明显，因此这并不是一个很难承受的负担。

在别的情况下，可能就不那么直观，就像在 3.2.4 节中你所看到的交换操作那样。至少在这种情况下，你可以同时锁定这些互斥元，但并不总是可能的。如果你回顾一下 3.1 节中链表的例子，你会看到一种保护链表的可能性，就是让每个结点都有一个互斥元。然后，为了访问这个链表，线程必须获取它们感兴趣的每个结点上的锁。对于一个删除某项的线程，它就必须获得三个结点上的锁，要删除的结点以及它两边的结点，因为它们全都要以某种方式进行修改。同样地，为了遍历链表，线程在获取序列中下一个结点上的锁的时候，必须保持当前结点上的锁，以确保指向下一结点的指针在此期间不被修改。一旦获取到下一个结点上的锁，就可以释放前面结点上的锁，因为它已经没用了。

这种逐节向上的锁定方式允许多线程访问链表，前提是每个线程访问不同的结点。然而，为了避免死锁，必须始终以相同的顺序锁定结点。如果两个线程试图用逐节锁定的方式以相反的顺序遍历链表，它们就会在链表中间产生相互死锁。如果结点 A 和 B 在链表中相邻，一个方向上的线程会试图保持锁定结点 A，并尝试获取结点 B 上的锁。而另一个方向上的线程会保持锁定结点 B，并且尝试获得结点 A 上的锁——死锁的典型情况。

同样地，当删除位于结点 A 和 C 之间的结点 B 时，如果该线程在获取结点 A 和 C 上的锁之前获取 B 上的锁，它就有可能与遍历链表的线程产生死锁。这样的线程会试图首先锁定 A 或 C（取决于遍历的方向），但是它接下来会发现无法获得结点 B 上的锁，因为正在进行删除操作的线程持有了结点 B 上的锁，并试图获得结点 A 和 C 上的锁。

在这里防止死锁的一个办法是定义遍历的顺序，让线程必须始终在锁定 B 之前锁定 A、在锁定 C 之前锁定 B。该方法以禁止反向遍历为代价来消除产生死锁的可能。对于其他数据结构，常常会建立类似的约定。

#### 4．使用锁层次

虽然这实际上是定义锁定顺序的一个特例，但锁层次能够提供一种方法，来检查在运行时是否遵循了约定。其思路是将应用程序分层，并且确认所有能够在任意给定的层

级上被锁定的互斥元。当代码试图锁定一个互斥元时，如果它在较低层已经持有锁定，那么就不允许它锁定该互斥元。通过给每一个互斥元分配层号，并记录下每个线程都锁定了哪些互斥元，你就可以在运行时进行检查了。清单 3.7 列出了两个线程使用层次互斥元的例子。

**清单 3.7 使用锁层次来避免死锁**

```
hierarchical_mutex high_level_mutex(10000);   ❶
hierarchical_mutex low_level_mutex(5000);     ❷

int do_low_level_stuff();

int low_level_func()
{
    std::lock_guard<hierarchical_mutex> lk(low_level_mutex);   ❸
    return do_low_level_stuff();
}

void high_level_stuff(int some_param);

void high_level_func()
{
    std::lock_guard<hierarchical_mutex> lk(high_level_mutex);   ❹
    high_level_stuff(low_level_func());   ❺
}

void thread_a()   ❻
{
    high_level_func();
}

hierarchical_mutex other_mutex(100);   ❼
void do_other_stuff();

void other_stuff()
{
    high_level_func();   ❽
    do_other_stuff();
}

void thread_b()   ❾
{
    std::lock_guard<hierarchical_mutex> lk(other_mutex);   ❿
    other_stuff();
}
```

thread_a() ❻ 遵守了规则，所以它运行良好。另一方面，thread_b() ❾无视了规则，因此将在运行时失败。thread_a()调用 high_level_func()，它锁定了 high_level_mutex❹（具有层次值 10000❶）并接着使用这个锁定了的互斥元调用 low_level_func()❺，以获得 high_level_stuff()的参数。low_level_func()接着锁定了 low_level_mutex❸，但是没关系，因为该互斥元具有较低的层次值 5000❷。

在另一方面 thread_b()却不妥。刚开始，它锁定了 other_mutex❿，它具有的层次值仅为 100❼。这意味着它应该是保护着超低级别的数据。当 other_stuff()调用 high_level_func()❽时，就会违反层次。high_level_func()试图获取值为 10000 的 high_level_mutex，大大超过 100 的当前层次值。因此，hierarchical_mutex 可能通过引发异常或终止程序来报错。层次互斥元之间的死锁是不可能出现的，因为互斥元本身实行了锁定顺序。这还意味着如果两个锁在层次中处于相同级别，你就不能同时持有它们，因此逐节锁定的方案要求链条中的每个互斥元具有比前一个互斥元更低的层次值，在某些情况下这可能是不切实际的。

这个例子也展现了另外一点，带有用户定义的互斥元类型的 std::lock_guard<>模板的使用。hierarchical_mutex 不是标准的一部分，但易于编写。清单 3.8 中展示了一个简单的实现。即便它是个用户定义的类型，但是可以用于 std::lock_guard<>，这是因为它实现了满足互斥元概念所需要的三个成员函数：lock()、unlock()和 try_lock()。你还没有见过直接使用 try_lock()，但它是相当简单的。如果互斥元上的锁已被另一个线程持有，则返回 false，而非一直等到调用线程可以获取该互斥元上的锁。try_lock()也可以在 std::lock()内部，作为避免死锁算法的一部分来使用。

**清单 3.8 简单的分层次互斥元**

```
class hierarchical_mutex
{
    std::mutex internal_mutex;
    unsigned long const hierarchy_value;
    unsigned long previous_hierarchy_value;
    static thread_local unsigned long this_thread_hierarchy_value;     <-❶
    void check_for_hierarchy_violation()
    {
        if(this_thread_hierarchy_value <= hierarchy_value)     <-❷
        {
            throw std::logic_error("mutex hierarchy violated");
        }
    }
    void update_hierarchy_value()
    {
        previous_hierarchy_value=this_thread_hierarchy_value;     <-❸
        this_thread_hierarchy_value=hierarchy_value;
    }
public:
    explicit hierarchical_mutex(unsigned long value):
        hierarchy_value(value),
        previous_hierarchy_value(0)
    {}
```

```
    void lock()
    {
        check_for_hierarchy_violation();
        internal_mutex.lock();                                    ④
        update_hierarchy_value();                                  ⑤
    }
    void unlock()
    {
        this_thread_hierarchy_value=previous_hierarchy_value;    ⑥
        internal_mutex.unlock();
    }
    bool try_lock()
    {
        check_for_hierarchy_violation();
        if(!internal_mutex.try_lock())                             ⑦
            return false;
        update_hierarchy_value();
        return true;
    }
};
thread_local unsigned long
    hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX);   ⑧
```

这里的关键是使用 thread_local 的值来表示当前线程的层次值：this_thread_hierarchy_value❶。它被初始化为最大值❽，所以在刚开始的时候任意互斥元都可以被锁定。由于它被声明为thread_local，每个线程都有属于自己的副本，所以在一个线程中该变量的状态，完全独立于从另一个线程中读取的该变量状态。参阅附录A，A.8节可以获得关于thread_local的更多信息。

因此，当线程第一次锁定 hierarchical_mutex 的实例时，this_thread_hierarchy_value的值为ULONG_MAX。就其本质而言，ULONG_MAX比其他任意值都大，所以通过了 check_for_hierarchy_violation()❷ 中的检查。在检查通过之后，lock()代理内部的互斥元用以实际锁定❹。一旦该锁定成功，就可以更新层次值❺。

现在如果在持有第一个 hierarchical_mutex 上的锁的同时，锁定另一个hierarchical_mutex，则 this_thread_hierarchy_value 的值反映的是第一个互斥元的层次值。为了通过检查❷，第二个互斥元的层次值必须小于已经持有的互斥元的层次值。

现在，保存当前线程之前的层次值是很重要的，这样才能在unlock()中恢复它❻；否则，你就无法再次锁定一个具有更高层次值的互斥元，即便该线程并没有持有任何锁。因为只有当你持有 internal_mutex 时才能保存之前的层次值❸，并在解锁该内部互斥元之前释放它❻，你可以安全地将其存储在 hierarchical_mutex 自身中，因为它被内部互斥元上的锁安全地保护。

try_lock()和 lock()工作原理相同，只是，如果在 internal_mutex 上调用

try_lock()失败❼，那么你就无法拥有这个锁，所以不能更新层次值，并且返回 false 而不是 true。

虽然检测是在运行时间检查，但它至少不依赖于时间——你不必去等待能够导致死锁出现的罕见情况发生。此外，需要以这种方式划分应用程序和互斥元的设计流程，可以在写入代码之前帮助消除许多可能导致死锁的原因。即使你还没有到达实际编写运行时间检测的那一步，进行设计练习仍然是值得的。

5. 将这些设计准则扩展到锁之外

正如我在本节开始时提到的，死锁不只是出现于锁定中，它可以发生在任何可以导致循环等待的同步结构中。因此，扩展上面所述的准则来涵盖那些情况也是值得的。举个例子，正如你应该尽量避免获取嵌套锁那样，在持有锁时等待一个线程是坏主意，因为该线程可能需要获取这个锁以继续运行。类似地，如果你正要等待一个线程完成，指定线程层次结构可能也是值得的，这样线程就只需要等待低层次上的线程。一个简单的做到这一点的方法，就是确保你的线程在启动它们的同一个函数中被结合，就像 3.1.2 节和 3.3 节中所描述的那样。

一旦你设计了代码来避免死锁，std::lock()和 std::lock_guard 涵盖了大多数简单锁定的情况，但有时却需要更大的灵活性。在那种情况下，标准库提供了 std::unique_lock 模板。与 std::lock_guard 类似，std::unique_lock 是在互斥元类型上进行参数化的类模板，并且它也提供了与 std::lock_guard 相同的 RAII 风格锁管理，但是更加灵活。

## 3.2.6 用 std::unique_lock 灵活锁定

通过松弛不变量，std::unique_lock 比 std::lock_guard 提供了更多的灵活性，一个 std::unique_lock 实例并不总是拥有与之相关联的互斥元。首先，就像你可以把 std::adopt_lock 作为第二参数传递给构造函数，以便让锁对象来管理互斥元上的锁那样，你也可以把 std::defer_lock 作为第二参数传递，来表示该互斥元在构造时应保持未被锁定。这个锁就可以在这之后通过在 std::unique_lock 对象（不是互斥元）上调用 lock()，或是通过将 std::unique_lock 对象本身传递给 std::lock()来获取。使用 std::unique_lock 和 std::defer_lock❶，而不是 std::lock_guard 和 std::adopt_lock，能够很容易地将清单 3.6 写成清单 3.9 中所示的那样。这段代码具有相同的行数，并且本质上是等效的，除了一个小问题，std::unique_lock 占用更多空间并且使用起来比 std::lock_guard 略慢。允许 std::unique_lock 实例不拥有互斥元的灵活性是有代价的，这条信息必须被存储，并且必须被更新。

**清单 3.9　在交换操作中使用 std::lock()和 std::unique_lock**

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);
class X
{
private:
    some_big_object some_detail;
    std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd){}

    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
            return;
        std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock);
        std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock);
        std::lock(lock_a,lock_b);
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

❶ std::defer_lock 保留互斥元为未锁定

❷ 互斥元在这里被锁定

在清单 3.9 中，std::unique_lock 对象能够被传递给 std::lock()❷，因为 std::unique_lock 提供了 lock()、try_lock()和 unlock()三个成员函数。它们会转发给底层互斥元上同名的成员函数去做实际的工作，并且只是更新在 std::unique_lock 实例内部的一个标识，来表示该实例当前是否拥有此互斥元。为了确保 unlock()在析构函数中被正确调用，这个标识是必需的。如果该实例确已拥有此互斥元，则析构函数**必须**调用 unlock()，并且，如果该实例**并未**拥有此互斥元，则析构函数**绝不能**调用 unlock()。可以通过调用 owns_lock()成员函数来查询这个标识。

如你所想，这个标识必须被存储在某个地方。因此，std::unique_lock 对象的大小通常大于 std::lock_guard 对象，并且相比于 std::lock_guard，使用 std::unique_lock 的时候，会有些许性能损失，因为需要对标识进行相应的更新或检查。如果 std::lock_guard 足以满足需求，我会建议优先使用它。也就是说，还有一些使用 std::unique_lock 更适合于手头任务的情况，因为你需要利用额外的灵活性。一个例子就是延迟锁定，正如你已经看到的；另一种情况是锁的所有权需要从一个作用域转移到另一个作用域。

### 3.2.7　在作用域之间转移锁的所有权

因为 std::unique_lock 实例并没有拥有与其相关的互斥元，所以通过四处移动（**moving**）实例，互斥元的所有权可以在实例之间进行转移。在某些情况下这种转移是自动的，比如从函数中返回一个实例，而在其他情况下，你必须通过调用 std::move()

来显式实现。从根本上说，这取决于源是否为**左值**（**lvalue**）——实变量或对实变量的引用——或者是**右值**（**rvalue**）——某种临时量。如果源为右值，则所有权转移是自动的，而对于左值，所有权转移必须显式地完成，以避免从变量中意外地转移了所有权。std::unique_lock 就是**可移动**（**movable**）但不可**复制**（**copyable**）的类型的例子。关于移动语义的详情，可参阅附录 A 中 A.1.1 节。

一种可能的用法，是允许函数锁定一个互斥元，并将此锁的所有权转移给调用者，于是调用者接下来可以在同一个锁的保护下执行额外的操作。下面的代码片段展示了这样的例子：函数 get_lock() 锁定了互斥元，然后在将锁返回给调用者之前准备数据。

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;                                    <-❶
}
void process_data()
{
    std::unique_lock<std::mutex> lk(get_lock());  <-❷
    do_something();
}
```

因为 lk 是在函数内声明的自动变量，它可以被直接返回❶而无需调用 std::move()；编译器负责调用移动构造函数。process_data() 函数可以直接将所有权转移到它自己的 std::unique_lock 实例❷，并且对 do_something() 的调用能够依赖被正确准备了的数据，而无需另一个线程在此期间去修改数据。

通常使用这种模式，是在待锁定的互斥元依赖于程序的当前状态，或者依赖于传递给返回 std::unique_lock 对象的函数的参数的地方。这种用法之一，就是并不直接返回锁，但是使用一个网关类的数据成员，以确保正确锁定了对受保护的数据的访问。这种情况下，所有对该数据的访问都通过这个网关类，当你想要访问数据时，就获取这个网关类的实例（通过调用类似于前面例子中的 get_lock() 函数），它会获取锁。然后，你可以通过网关对象的成员函数来访问数据。在完成后，销毁网关对象，从而释放锁，并允许其他线程访问受保护的数据。这样的网关对象很可能是可移动的（因此它可以从函数返回），在这种情况下，锁对象的数据成员也需要是可移动的。

std::unique_lock 的灵活性同样允许实例在被销毁之前撤回它们的锁。你可以使用 unlock() 成员函数来实现，就像对于互斥元那样，std::unique_lock 支持与互斥元一样的用来锁定和解锁的基本成员函数集合，这是为了让它可以用于通用函数，比如 std::lock。在 std::unique_lock 实例被销毁之前释放锁的能力，意味着你可以有选择地在特定的代码分支释放锁，如果很显然不再需要这个锁，这对于应用程序的性能可能很重要。持有锁的时间比所需时间更长，会导致性能下降，因为其他等待该

锁的线程，被阻止运行超过了所需的时间。

### 3.2.8 锁定在恰当的粒度

锁粒度是我在之前曾提到过的，在 3.2.3 节中：锁粒度是一个文字术语，用来描述由单个锁所保护的数据量。细粒度锁保护着少量的数据，粗粒度锁保护着的大量的数据。选择一个足够粗的锁粒度，来确保所需的数据都被保护是很重要的，不仅如此，同样重要的是，确保只在真正需要锁的操作中持有锁。我们都知道，带着满满一车杂货在超市排队结账，只因为正在结账的人突然意识到自己忘了一些小红莓酱，然后就跑去找，而让大家都等着，或者收银员已经准备好收钱，顾客才开始在自己的手提包里翻找钱包，是很令人抓狂的。如果每个人去结账时都拿到了他们想要的，并准备好了适当的支付方式，一切都更容易进行。

这同样适用于线程，如果多个线程正等待着同一个资源（收银台的收银员），然后，如果任意线程持有锁的时间比所需时间长，就会增加等待所花费的总时间（不要等到你已经到了收银台才开始寻找小红莓酱）。如果可能，仅在实际访问共享数据的时候锁定互斥元，尝试在锁的外面做任意的数据处理。特别地，在持有锁时，不要做任何确实很耗时的活动，比如文件 I/O。文件 I/O 通常比从内存中读取或写入相同大小的数据量要慢上数百倍（如果不是数千倍）。因此，除非这个锁是真的想保护对文件的访问，否则在持有锁时进行 I/O 会不必要地延迟其他线程（因为它们在等待获取锁时会阻塞），潜在地消除了使用多线程带来的性能提升。

std::unique_lock 在这种情况下运作良好，因为能够在代码不再需要访问共享数据时调用 unlock()，然后在代码中又需要访问时再次调用 lock()。

```
void get_and_process_data()
{
    std::unique_lock<std::mutex> my_lock(the_mutex);
    some_class data_to_process=get_next_data_chunk();
    my_lock.unlock();                                   ❶ 在对 process()的调用中不需要锁定互斥元
    result_type result=process(data_to_process);
    my_lock.lock();                                     ❷ 重新锁定互斥元以回写结果
    write_result(data_to_process,result);
}
```

在调用 process()过程中不需要锁定互斥元，所以手动地将其在调用前解锁❶，并在之后再次锁定❷。

希望这是显而易见的，如果你让一个互斥锁保护整个数据结构，不仅可能会有更多的对锁的竞争，锁被持有的时间也可能会减少。更多的操作步骤会需要在同一个互斥元上的锁，所以锁必须被持有更长的时间。这种成本上的双重打击，也是尽可能走向细粒度锁定的双重激励。

如这个例子所示，锁定在恰当的粒度不仅关乎锁定的数据量；这也是关系到锁会被

持有多长时间，以及在持有锁时执行哪些操作。一般情况下，只应该以执行要求的操作所需的最小可能时间而去持有锁。这也意味着耗时的操作，比如获取另一个锁（即便你知道它不会死锁）或是等待 I/O 完成，都不应该在持有锁的时候去做，除非绝对必要。

在清单 3.6 和清单 3.9 中，需要锁定两个互斥锁的操作是交换操作，这显然需要并发访问两个对象。假设取而代之，你试图去比较仅为普通 int 的简单数据成员。这会有区别吗？int 可以轻易被复制，所以你可以很容易地为每个待比较的的对象复制其数据，同时只用持有该对象的锁，然后比较已复制数值。这意味着你在每个互斥元上持有锁的时间最短，并且你也没有在持有一个锁的时候去锁定另外一个。清单 3.10 展示了这样的一个类 Y，以及相等比较运算符的示例实现。

**清单 3.10 在比较运算符中每次锁定一个互斥元**

```
class Y
{
private:
    int some_detail;
    mutable std::mutex m;
    int get_detail() const
    {
        std::lock_guard<std::mutex> lock_a(m);      ←❶
        return some_detail;
    }
public:
    Y(int sd):some_detail(sd){}

    friend bool operator==(Y const& lhs, Y const& rhs)
    {
        if(&lhs==&rhs)
            return true;
        int const lhs_value=lhs.get_detail();              ←❷
        int const rhs_value=rhs.get_detail();          ←❸
        return lhs_value==rhs_value;           ←❹
    }
};
```

在这种情况下，比较运算符首先通过调用 get_detail()成员函数获取要进行比较的值❷、❸。此函数在获取值的同时用一个锁来保护它❶。比较运算符接着比较获取到的值❹。但是请注意，这同样会减少锁定的时间，而且每次只持有一个锁（从而消除了死锁的可能性），与同时持有两个锁相比，**这巧妙地改变了操作的语义**。在清单 3.10 中，如果运算符返回 true，意味着 lhs.some_detail 在一个时间点的值与 rhs.some_detail 在另一个时间点的值相等。这两个值能够在两次读取之中以任何方式改变。例如，这两个值可能在❷和❸之间进行了交换，从而使这个比较变得毫无意义。这个相等比较可能会返回 true 来表示值是相等的，即使这两个值在某个瞬间从未

真正地相等过。因此，当进行这样的改变时小心注意是很重要的，操作的语义不能以有问题的方式而被改变：如果你不能在操作的整个持续时间中持有所需的锁，你就把自己暴露在竞争条件中。

有时，根本就没有一个合适的粒度级别，因为并非所有的对数据结构的访问都要求同样级别的保护。在这种情况下，使用替代机制来代替普通的 std::mutex 可能才是恰当的。

## 3.3 用于共享数据保护的替代工具

虽然互斥元是最通用的机制，但提到保护共享数据时，它们并不是唯一的选择；还有别的替代品，可以在特定情况下提供更恰当的保护。

一个特别极端（但却相当常见）的情况，就是共享数据只在初始化时才需要并发访问的保护，但在那之后却不需要显式同步。这可能是因为数据是一经创建就是只读的，所以就不存在可能的同步问题，或者是因为必要的保护作为数据上操作的一部分被隐式地执行。在任一情况中，在数据被初始化之后锁定互斥元，纯粹是为了保护初始化，这是不必要的，并且对性能会产生的不必要的打击。为了这个原因，C++标准提供了一种机制，纯粹为了在初始化过程中保护共享数据。

### 3.3.1 在初始化时保护共享数据

假设你有一个构造起来非常昂贵的共享资源，只有当实际需要时你才会要这样做。也许，它会打开一个数据库连接或分配大量的内存。像这样的延迟初始化（**lazy initialization**）在单线程代码中是很常见的——每个请求资源的操作首先检查它是否已经初始化，如果没有就在使用之前初始化之。

```
std::shared_ptr<some_resource> resource_ptr;
void foo()
{
    if(!resource_ptr)
    {
        resource_ptr.reset(new some_resource);      <-❶
    }
    resource_ptr->do_something();
}
```

如果共享资源本身对于并发访问是安全的，当将其转换为多线程代码时唯一需要保护的部分就是初始化❶，但是像清单 3.11 中这样的朴素的转换，会引起使用该资源的线程产生不必要的序列化。这是因为每个线程都必须等待互斥元，以检查资源是否已经被初始化。

**清单 3.11 使用互斥元进行线程安全的延迟初始化**

```
std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;
void foo()
{
    std::unique_lock<std::mutex> lk(resource_mutex);    <-- 所有的线程在这里被序列化
    if(!resource_ptr)
    {
        resource_ptr.reset(new some_resource);    <-- 只有初始化需要被保护
    }
    lk.unlock();
    resource_ptr->do_something();
}
```

这段代码是很常见的，不必要的序列化问题已足够大，以至于许多人都试图想出一个更好的方法来实现，包括臭名昭著的二次检查锁定（**Double-Checked Locking**）模式，在不获取锁❶（在下面的代码中）的情况下首次读取指针，并仅当此指针为 NULL 时获得该锁。一旦已经获取了锁，该指针要被再次检查❷（这就是二次检查的部分），以防止在首次检查和这个线程获取锁之间，另一个线程就已经完成了初始化。

```
void undefined_behaviour_with_double_checked_locking()
{
    if(!resource_ptr)                                   <-- ❶
    {
        std::lock_guard<std::mutex> lk(resource_mutex);
        if(!resource_ptr)                               <-- ❷
        {
            resource_ptr.reset(new some_resource);      <-- ❸
        }
    }
    resource_ptr->do_something();                       <-- ❹
}
```

不幸的是，这种模式因某个原因而臭名昭著。它有可能产生恶劣的竞争条件，因为在锁外部的读取❶与锁内部由另一线程完成的写入不同步 ❸。这就因此创建了一个竞争条件，不仅涵盖了指针本身，还涵盖了指向的对象。就算一个线程看到另一个线程写入的指针，它也可能无法看到新创建的 some_resource 实例，从而导致 do_something() ❹ 的调用在不正确的值上运行。这是一个竞争条件的例子，该类型的竞争条件被 C++标准定义为**数据竞争**（**data race**），因此被定为**未定义行为**。因此，这是肯定需要避免的。内存模型的详细讨论参见第 5 章，包括了什么构成数据竞争。

C++ 标准委员会也发现这是一个重要的场景，所以 C++ 标准库提供了 std::once_flag 和 std::call_once 来处理这种情况。与其锁定互斥元并且显式地检查指针，还不如每个线程都可以使用 std::call_once，到 std::call_once 返回时，指针将会被某个线程初始化（以完全同步的方式），这样就安全了。使用 std::call_once 比显式使用互斥元通常会有更低的开销，特别是初始化已经完成的

时候，所以在 std::call_once 符合所要求的功能时应优先使用之。下面的例子展示了与清单 3.11 相同的操作，改写为使用 std::call_once。在这种情况下，通过调用函数来完成初始化，但是通过一个带有函数调用操作符的类实例也可以很容易地完成初始化。与标准库中接受函数或者断言作为参数的大部分函数类似，std::call_once 可以与任意函数或可调用对象合作。

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;                    <-❶

void init_resource()
{
    resource_ptr.reset(new some_resource);
}

void foo()
{
    std::call_once(resource_flag,init_resource);   <-- 初始化会被正好调用一次
    resource_ptr->do_something();
}
```

在这个例子中，std::once_flag❶ 和被初始化的数据都是命名空间作用域的对象，但是 std::call_once() 可以容易地用于类成员的延迟初始化，如清单 3.12 所示。

**清单 3.12　使用 std::call_once 的线程安全的类成员延迟初始化**

```
class X
{
private:
    connection_info connection_details;
    connection_handle connection;
    std::once_flag connection_init_flag;

    void open_connection()
    {
        connection=connection_manager.open(connection_details);
    }
public:
    X(connection_info const& connection_details_):
        connection_details(connection_details_)
    {}
    void send_data(data_packet const& data)          <-❶
    {
        std::call_once(connection_init_flag,&X::open_connection,this);   <-❷
        connection.send_data(data);
    }
    data_packet receive_data()                    <-❸
    {
        std::call_once(connection_init_flag,&X::open_connection,this);   <-❷
        return connection.receive_data();
    }
};
```

在这个例子中，初始化由首次调用 send_data()❶或是由首次调用 receive_data()❸来完成。使用成员函数 open_connection()来初始化数据，同样需要将 this 指针传入函数。和标准库中其他接受可调用对象的函数一样，比如 std::thread 和 std::bind()的构造函数，这是通过传递一个额外的参数给 std::call_once()来完成的❷。

值得注意的是，像 std::mutex、std::once_flag 的实例是不能被复制或移动的，所以如果想要像这样把它们作为类成员来使用，就必须显式定义这些你所需要的特殊成员函数。

一个在初始化过程中可能会有竞争条件的场景，是将局部变量声明为 static 的。这种变量的初始化，被定义为在时间控制首次经过其声明时发生。对于多个调用该函数的线程，这意味着可能会有针对定义"首次"的竞争条件。在许多 C++11 之前的编译器上，这个竞争条件在实践中是有问题的，因为多个线程可能都认为它们是第一个，并试图去初始化该变量，又或者线程可能会在初始化已在另一个线程上启动但尚未完成之时试图使用它。在 C++11 中，这个问题得到了解决。初始化被定义为只发生在一个线程上，并且其他线程不可以继续直到初始化完成，所以竞争条件仅仅在于哪个线程会执行初始化，而不会有更多别的问题。对于需要单一全局实例的场合，这可以用作 std::call_once 的替代品。

```
class my_class;
my_class& get_my_class_instance()
{
    static my_class instance;      <-- ❶ 初始化保证线程是安全的
    return instance;
}
```

多个线程可以继续安全地调用 get_my_class_instance()❶，而不必担心初始化时的竞争条件。

保护仅用于初始化的数据是更普遍的场景下的一个特例，那些很少更新的数据结构。对于大多数时间而言，这样的数据结构是只读的，因而可以毫无顾忌地被多个线程同时读取，但是数据结构偶尔可能需要更新。这里我们所需要的是一种承认这一事实的保护机制。

## 3.3.2 保护很少更新的数据结构

假设有一个用于存储 DNS 条目缓存的表，它用来将域名解析为相应的 IP 地址。通常，一个给定的 DNS 条目将在很长一段时间里保持不变——在许多情况下，DNS 条目会保持数年不变。虽然随着用户访问不同的网站，新的条目可能会被不时地添加到表中，但这一数据却将在其整个生命中基本保持不变。定期检查缓存条目的有效性是很重要的，但是只有在细节已有实际改变的时候才会需要更新。

虽然更新是罕见的，但它们仍然会发生，并且如果这个缓存可以从多个线程访问，它就需要在更新过程中进行适当的保护，以确保所有线程在读取缓存时都不会看到损坏的数据结构。

在缺乏完全符合预期用法并且为并发更新与读取专门设计（例如在第6章和第7章的那些）的专用数据结构的情况下，这种更新要求线程在进行更新时独占访问数据结构，直到它完成了操作。一旦更新完成，该数据结构对于多线程并发访问又是安全的了。使用 std::mutex 来保护数据结构就因而显得过于悲观，因为这会在数据结构没有进行修改时消除并发读取数据结构的可能，我们需要的是另一种互斥元。这种新的互斥元通常称为**读写**（**reader-writer**）互斥元，因为它考虑到了两种不同的用法：由单个“写”线程独占访问或共享，由多个“读”线程并发访问。

新的 C++标准库并没有直接提供这样的互斥元，尽管已向标准委员会提议[1]。由于这个建议未被接纳，本节中的例子使用由 Boost 库提供的实现，它是基于这个建议的。在第8章中你会看到，使用这样的互斥元并不是万能药，性能依赖于处理器的数量以及读线程和更新线程的相对工作负载。因此，分析代码在目标系统上的性能是很重要的，以确保额外的复杂度会有实际的收益。

你可以使用 boost::shared_mutex 的实例来实现同步，而不是使用 std::mutex 的实例。对于更新操作，std::lock_guard<boost::shared_mutex> 和 std::unique_lock<boost::shared_mutex> 可用于锁定，以取代相应的 std::mutex 特化。这确保了独占访问，就像 std::mutex 那样。那些不需要更新数据结构的线程能够转而使用 boost::shared_lock<boost::shared_mutex>来获得**共享**访问。这与 std::unique_lock 用起来正是相同的，除了多个线程在同一时间、同一 boost::share_mutex 上可能会具有共享锁。唯一的限制是，如果任意一个线程拥有一个共享锁，试图获取独占锁的线程会被阻塞，直到其他线程全都撤回它们的锁，同样地，如果任意一个线程具有独占锁，其他线程都不能获取共享锁或独占锁，直到第一个线程撤回了它的锁。

清单 3.13 展示了一个简单的如前面所描述的 DNS 缓存，使用 std::map 来保存缓存数据，用 boost::share_mutex 进行保护。

**清单 3.13 使用 boost::share_mutex 保护数据结构**

```
#include <map>
#include <string>
#include <mutex>
#include <boost/thread/shared_mutex.hpp>
```

---

1 Howard E. Hinnant, “Multithreading API for C++0X—A Layered Approach”, C++ Standards Committee Paper N2094, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2094.html

```
class dns_entry;

class dns_cache
{
    std::map<std::string,dns_entry> entries;
    mutable boost::shared_mutex entry_mutex;
public:
    dns_entry find_entry(std::string const& domain) const
    {
        boost::shared_lock<boost::shared_mutex> lk(entry_mutex);    <-❶
        std::map<std::string,dns_entry>::const_iterator const it=
            entries.find(domain);
        return (it==entries.end())?dns_entry():it->second;
    }
    void update_or_add_entry(std::string const& domain,
                             dns_entry const& dns_details)
    {
        std::lock_guard<boost::shared_mutex> lk(entry_mutex);    <-❷
        entries[domain]=dns_details;
    }
};
```

在清单 3.13 中，find_entry()使用一个 boost::share_lock<>实例来保护它，以供共享、只读的访问 ❶；多个线程因而可以毫无问题地同时调用 find_entry()。另一方面，update_or_add_entry()使用一个 std::lock_guard<>实例，在表被更新时提供独占访问 ❷；不仅在调用 update_or_add_entry()中其他线程被阻止进行更新，调用 find_entry()的线程也会被阻塞。

### 3.3.3 递归锁

在使用 std::mutex 的情况下，一个线程试图锁定其已经拥有的互斥元是错误的，并且试图这么做将导致未定义行为（**undefined behavior**）。然而，在某些情况下，线程多次重新获取同一个互斥元却无需事先释放它是可取的。为了这个目的，C++标准库提供了 std::recursive_mutex。它就像 std::mutex 一样，区别在于你可以在同一个线程中的单个实例上获取多个锁。在互斥元能够被另一个线程锁定之前，你必须释放所有的锁，因此如果你调用 lock()三次，你必须也调用 unlock()三次。正确使用 std::lock_guard<std::recursive_mutex> 和 std::unique_lock<std::recursive_mutex>将会为你处理。

大多数时间，如果你觉得需要一个递归互斥元，你可能反而需要改变你的设计。递归互斥元常用在一个类被设计成多个线程并发访问的情况中，因此它具有一个互斥元来保护成员数据。每个公共成员函数锁定互斥元，进行工作，然后解锁互斥元。然而，有时一个公共成员函数调用另一个函数作为其操作的一部分是可取的。在这种情况下，第二个成员函数也将尝试锁定该互斥元，从而导致未定义行为。粗制滥造的解决方案，就

是将互斥元改为递归互斥元。这将允许在第二个成员函数中对互斥元的锁定成功进行，并且函数继续。

然而，这样的用法是**不推荐**的，因为它可能导致草率的想法和糟糕的设计。特别地，类的不变量在锁被持有时通常是损坏的，这意味着第二个成员函数需要工作，即便在被调用时使用的是损坏的不变量。通常最好是提取一个新的私有成员函数，该函数是从这两个成员函数中调用的，它不锁定互斥元（它认为互斥元已经被锁定)。然后，你可以仔细想想在什么情况下可以调用这个新函数以及在那些情况下数据的状态。

## 3.4 小结

在本章中，我讨论了在线程之间共享数据时，有问题的竞争条件如何成为灾难，以及怎样使用 std:mutex 和精心设计接口以避免它们。你看到了互斥元不是万能药，也有它们自己的以死锁形式出现的问题，尽管 C++标准库以 std::lock()的形式提供了工具来帮助避免死锁。然后，你看到了一些进一步的技术来避免死锁，接着简要看了一下锁的所有权的转让，以围绕着为锁选择恰当的粒度的问题。最后，我介绍了为特定场景提供的替代的数据保护工具，例如 std::call_once()和 boost::shared_mutex。

然而，还有一件事我没有提到，就是等待来自其他线程的输入。我们的线程安全栈在栈为空的情况下只是引发异常，因此如果一个线程需要等待另一个线程来将一个值压入栈中（毕竟，这是线程安全栈的主要用途之一)，它将不得不反复尝试弹出值，如果引发异常则重试。这会消耗宝贵的处理时间来进行检查，而没有实际取得任何进展。的确，不断地检查可能会通过阻止系统中其他线程的运行而**阻碍**进度。我们需要的是以某种方法让一个线程等待另一个线程完成任务，而无需耗费 CPU 时间。第 4 章构建在已经讨论过的用于保护共享数据的工具上，介绍了 C++中用于线程间同步操作的各种机制，第 6 章展示了如何使用它们来构建更大的可复用的数据结构。

# 第 4 章 同步并发操作

本章主要内容

- 等待事件
- 使用 future 来等待一次性事件
- 有时间限制的等待
- 使用操作的同步来简化代码

在上一章中，我们看到了各种方法去保护在线程间共享的数据。但是有时候你不只是需要保护数据，还需要在独立的线程上进行同步操作。例如，一个线程在能够完成其任务之前可能需要等待另一个线程完成任务。一般来说，希望一个线程等待特定事件的发生或是一个条件变为 true 是常见的事情。虽然通过定期检查“任务完成”的标识或是在共享数据中存储类似的东西也能够做到这一点，但却不甚理想。对于像这样的线程间同步操作的需求是如此常见，以至于 C++标准库提供了以条件变量（**condition variables**）和期值（**future**）为形式的工具来处理它。

在本章中，我将讨论如何使用条件变量和期值来等待事件，以及如何使用它们来简化操作的同步。

## 4.1 等待事件或其他条件

假设你正乘坐通宵列车旅行。一个可以确保你在正确的车站下车的方法就是整夜保

持清醒并注意火车停靠的地方。你不会误站，但你到那儿时就会觉得很累。或者，你可以查一下时间表，了解火车会在何时到达，将闹钟定的稍微提前一点，然后去睡觉。这是可以的；你不会错过站，但是如果火车晚点了，你就会醒得太早。也有可能闹钟的电池没电了，你就会睡过头以至于错过站。理想的状况是，你只管去睡觉，让某个人或某个东西在火车到站时叫醒你，无论何时。

这如何与线程相关呢？那么，如果一个线程正等待着第二个线程完成一项任务，它有几个选择。首先，它可以一直检查共享数据（由互斥元保护）中的标识，并且让第二个线程在完成任务时设置该标识。这有两项浪费，线程占用了宝贵的处理时间去反复检查该标识，以及当互斥元被等待的线程锁定后，就不能被任何其他线程锁定。两者都反对线程进行等待，因为它们限制了等待中的线程的可用资源，甚至阻止它在完成任务时设置标识。这类似于整夜保持清醒地与火车司机交谈，他不得不把火车开得更慢，因为你一直在干扰他，所以需要更长的时间才能到达。同样的，等待中的线程消耗了本可以被系统中其他线程使用的资源，并且最终等待的时间可能会比所需的更长。

第二个选择是使用 std::this_thread::sleep_for()函数（参见 4.3 节），让等待中的线程在检查之间休眠一会儿。

```
bool flag;
std::mutex m;

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();        // ❶ 解锁互斥元
        std::this_thread::sleep_for(std::chrono::milliseconds(100));   // ❷ 休眠 100 毫秒
        lk.lock();          // ❸ 重新锁定互斥元
    }
}
```

在这个循环里，函数在休眠之前 ❷ 解锁该互斥元 ❶，并在之后再次锁定之 ❸，所以另一个线程有机会获取它并设置标识。

这是一个进步，因为线程在休眠时并不浪费处理时间，但得到正确的休眠时间是很难的。检查之间休眠得过短，线程仍然会浪费处理时间进行检查；休眠得过长，即使线程正在等待的任务已经完成，它还会继续休眠，导致延迟。这种过度休眠很少直接影响程序的操作，但它可能意味着在快节奏的游戏中丢帧，或者在实时应用程序中过度运行一个时间片。

第三个选择，同时也是首选选择，是使用 C++标准库提供的工具来等待事件本身。等待由另一个线程触发一个事件的最基本机制（例如前面提到的管道中存在的额外操作）是**条件变量（condition variable）**。从概念上说，条件变量与某些事件或其他条件相关，并且一个或多个线程可以等待该条件被满足。当某个线程已经确定条件得到满足，它就可以通

知一个或多个正在条件变量上进行等待的线程，以便唤醒它们并让它们继续处理。

### 4.1.1 用条件变量等待条件

标准 C++库提供了两个条件变量的实现：std::condition_variable 和 std::condition_variable_any。这两个实现都在<condition_variable>库的头文件中声明。两者都需要和互斥元一起工作，以便提供恰当的同步；前者仅限于和 std::mutex 一起工作，而后者则可以与符合成为类似互斥元的最低标准的任何东西一起工作，因此以 any 为后缀。因为 std::condition_variable_any 更加普遍，所以会有大小、性能或者操作系统资源方面的形式的额外代价的可能，因此应该首选 std::condition_variable，除非需要额外的灵活性。

那么，如何使用 std::condition_variable 去处理引言中的例子——怎么让正在等待工作的线程休眠，直到有数据要处理？清单 4.1 展示了一种方法，你可以用条件变量来实现这一点。

**清单 4.1 使用 std::condition_variable 等待数据**

```
std::mutex mut;
std::queue<data_chunk> data_queue;   <-❶
std::condition_variable data_cond;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);          <-❷
        data_cond.notify_one();        <-❸
    }
}
void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);       <-❹
        data_cond.wait(
            lk,[]{return !data_queue.empty();});    <-❺
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();          <-❻
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

首先，你拥有一个用来在两个线程之间传递数据的队列 ❶。当数据就绪时，准备数据的线程使用 std::lock_guard 去锁定保护队列的互斥元，并且将数据压入队列中 ❷。然后，它在 std::condition_variable 的实例上调用 notify_one()成员函数，以通知等待中的线程（如果有的话）❸。

在另外一侧，你还有处理线程。该线程首先锁定互斥元，但是这次使用的是 std::unique_lock 而不是 std::lock_guard❹——你很快就会明白为什么。该线程接下来在 std::condition_variable 上调用 wait()，传入锁对象以及表示正在等待的条件的 lambda 函数 ❺。lambda 函数是 C++中的一个新功能，它允许你编写一个匿名函数作为另一个表达式的一部分，它们非常适合于为类似于 wait()这样的标准库函数指定断言。在这个例子中，简单的lambda 函数[]{return !data_queue.empty();}检查 data_queue 是否不为 empty()，即队列中已有数据准备处理。附录 A，A.5 节更加详细地描述了 lambda 函数。

wait()的实现接下来检查条件（通过调用所提供的 lambda 函数），并在满足时返回（lambda 函数返回 true）。如果条件不满足（lambda 函数返回 false），wait()解锁互斥元，并将该线程置于阻塞或等待状态。当来自数据准备线程中对 notify_one()的调用通知条件变量时，线程从睡眠状态中苏醒（解除其阻塞），重新获得互斥元上的锁，并再次检查条件，如果条件已经满足，就从 wait()返回值，互斥元仍被锁定。如果条件不满足，该线程解锁互斥元，并恢复等待。这就是为什么需要 std::unique_lock 而不是 std:: lock_guard——等待中的线程在等待期间必须解锁互斥元，并在这之后重新将其锁定，而 std::lock_guard 没有提供这样的灵活性。如果互斥元在线程休眠期间始终被锁定，数据准备线程将无法锁定该互斥元，以便将项目添加至队列，并且等待中的线程将永远无法看到其条件得到满足。

清单 4.1 为等待使用了一个简单的 lambda 函数 ❺，该函数检查队列是否为非空的，但是任何函数或可调用对象都可以传入。如果你已经有一个函数来检查条件（也许因为它比这样一个简单的试验更加复杂），那么这个函数就可以直接传入，没有必要将其封装在 lambda 中。在对 wait()的调用中，条件变量可能会对所提供的条件检查任意多次。然而，它总是在互斥元被锁定的情况下这样做，并且当（且仅当）用来测试条件的函数返回 true，它就会立即返回。当等待线程重新获取互斥元并检查条件时，如果它并非直接响应另一个线程的通知，这就是所谓的**伪唤醒**（**spurious wake**）。由于所有的这种伪唤醒的次数和频率根据定义是不确定的，所以使用对于条件检查具有副作用的函数是不可取的。如果你这样做，就必须做好多次产生副作用的准备。

解锁 std::unique_lock 的灵活性不仅适用于对 wait()的调用；它还可用于你有待处理但仍未处理的数据 ❻。处理数据可能是一个耗时的操作，并且如你在第 3 章中看到的，在互斥元上持有锁超过所需的时间就是个不好的情况。

清单 4.1 所示的使用队列在线程之间传输数据，是很常见的场景。做得好的话，同步可以被限制在队列本身，大大减少了同步问题和竞争条件大概的数量。鉴于此，现在让我们从清单 4.1 中提取一个泛型的线程安全队列。

### 4.1.2 使用条件变量建立一个线程安全队列

如果你要设计一个泛型队列，花几分钟考虑一下可能需要的操作是值得的，就像你在 3.2.3 节对线程安全堆栈所做的那样。让我们看一看 C++标准库来寻找灵感，以清单 4.2 所示的 std::queue<>的容器适配器的形式。

**清单 4.2 std::queue 接口**

```
template <class T, class Container = std::deque<T> >
class queue {
public:
    explicit queue(const Container&);
    explicit queue(Container&& = Container());

    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);

    void swap(queue& q);

    bool empty() const;
    size_type size() const;

    T& front();
    const T& front() const;
    T& back();
    const T& back() const;

    void push(const T& x);
    void push(T&& x);

    void pop();
    template <class... Args> void emplace(Args&&... args);
};
```

如果忽略构造函数、赋值和交换操作，那么还剩下 3 组操作：查询整个队列的状态（empty()和 size()）、查询队列的元素（front()和 back()）以及修改队列（push()、pop()和 emplace()）。这些操作与你之前在 3.2.3 节中对堆栈的操作是相同的，因此你也遇到相同的有关接口中固有的竞争条件的问题。所以，你需要将 front()和 pop()组合到单个函数调用中，就像你为了堆栈而组合 top()和 pop()那样。清单 4.1 中的代码增加了新的细微差别，但是，当使用队列在线程间传递数据时，接收线程往往需要等待数据。我们为 pop()提供了两个变体：try_pop()，它试图从队列中弹出值，但总是立即返回（带有失败指示符），即使没有能获取到值。以及

wait_and_pop()，它会一直等待，直到有值要获取。如果将栈示例中的特征带到此处，则接口看起来如清单 4.3 所示。

**清单 4.3　threadsafe_queue 的接口**

```
#include <memory>                    <-- 为了 std::shared_ptr

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(
        const threadsafe_queue&) = delete;    <-- 为简单起见不允许赋值

    void push(T new_value);

    bool try_pop(T& value);              <-- ❶
    std::shared_ptr<T> try_pop();        <-- ❷

    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();

    bool empty() const;
};
```

就像你为堆栈做的那样，减少构造函数并消除赋值以简化代码。如以前一样，还提供了 try_pop()和 wait_and_pop()的两个版本。try_pop()的第一个重载 ❶ 将获取到的值存储在引用变量中，所以它可以将返回值用作状态，如果它获取到值就返回 true，否则返回 false（参见 A.2 节）。第二个重载 ❷ 不能这么做，因为它直接返回获取到的值。但是如果没有值可被获取，则返回的指针可以设置为 NULL。

那么，所有这一切如何与清单 4.1 关联起来呢？嗯，你可以从那里提取 push()以及 wait_and_pop()的代码，如清单 4.4 所示。

**清单 4.4　从清单 4.1 中提取 push()和 wait_and_pop()**

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    void push(T new_value)
    {
```

```
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
};
threadsafe_queue<data_chunk> data_queue;    <-❶
void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        data_queue.push(data);              <-❷
    }
}
void data_processing_thread()
{
    while(true)
    {
        data_chunk data;
        data_queue.wait_and_pop(data);     <-❸
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

互斥元和条件变量现在包含在 threadsafe_queue 的实例中，所以不再需要单独的变量 ❶，并且调用 push()不再需要外部的同步 ❷。此外，wait_and_pop()负责条件变量等待 ❸。

wait_and_pop()的另一个重载现在很容易编写，其余的函数几乎可以一字不差地从清单 3.5 中的栈示例中复制过来。清单 4.5 展示了最终的队列实现。

**清单 4.5 使用条件变量的线程安全队列的完整类定义**

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
```

```
{
private:
    mutable std::mutex mut;         <-- ❶ 互斥元必须是
    std::queue<T> data_queue;           可变的
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=data_queue.front();
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
```

```
        return res;
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

虽然 empty()是一个 const 成员函数，并且拷贝构造函数的 other 参数是一个 const 引用，但是其他线程可能会有到该对象的非 const 引用，并调用可变的成员函数，所以我们仍然需要锁定互斥元。由于锁定互斥元是一种可变的操作，故互斥元对象必须标记为 mutable❶，以便其可以被锁定在 empty()和拷贝构造函数中。

条件变量在多个线程等待同一个事件的场合也是很有用的。如果线程被用于划分工作负载，那么应该只有一个线程去响应通知，可以使用与清单 4.1 中所示完全相同的结构，只要运行多个数据处理线程的实例。当新的数据准备就绪，notify_one()的调用将触发其中一个正在执行 wait()的线程去检查其条件，然后从 wait()返回（因为你刚向 data_queue 中增加了一项）。谁也不能保证哪个线程会被通知到，即使是某个线程正在等待通知；所有的处理线程可能仍在处理数据。

另一种可能性是，多个线程正等待着同一个事件，并且它们都需要作出响应。这可能发生在共享数据正在初始化的场合下，处理线程都可以使用同一个数据，但需要等待其初始化完成（虽然有比这更好的机制，参见第 3 章中的 3.3.1 节），或者是线程需要等待共享数据更新的地方，比如周期性的重新初始化。在这些案例中，准备数据的线程可以在条件变量上调用 notify_all()成员函数而不是 notify_one()。顾名思义，这将导致所有当前执行着 wait()的线程检查其等待中的条件。

如果等待线程只打算等待一次，那么当条件为 true 时它就不会再等待这个条件变量了，条件变量未必是同步机制的最佳选择。如果所等待的条件是一个特定数据块的可用性时，这尤其正确。在这个场景中，使用期值（**future**）可能会更合适。

## 4.2 使用 future 等待一次性事件

假设你要乘飞机去国外度假。在到达机场并且完成了各种值机手续后，你仍然需要等待航班准备登机的通知，也许要几个小时。当然，你可以找到一些方法来消磨时间，比如看书、上网或者在昂贵的机场咖啡厅吃东西，但是从根本上来说，你只是在等待一件事情：登机时间到了的信号。不仅如此，一个给定的航班只进行一次；你下次去度假的时候，就会等待不同的航班。

C++标准库使用 **future** 为这类一次性事件建模。如果一个线程需要等待特定的一次性事件，那么它就会获取一个 future 来代表这一事件。然后，该线程可以周期性地在这

个 future 上等待一小段时间以检查事件是否发生（检查出发告示板），而在检查间隙执行其他的任务（在高价咖啡厅吃东西）。另外，它还可以去做另外一个任务，直到其所需的事件已发生才继续进行，随后就等待 future 变为**就绪**（**ready**）。future 可能会有与之相关的数据（比如你的航班在哪个登机口登机），或可能没有。一旦事件已经发生（即 future 已变为**就绪**），future 就无法复位。

C++标准库中有两类 future，是由`<future>`库的头文件中声明的两个类模板实现的：**唯一 future**（unique futures，`std::future<>`）和**共享 future**（shared futures，`std::shared_future<>`）。这两个类模板是参照 `std::unique_ptr` 和 `std::shared_ptr` 建立的。`std::future`的实例是仅有的一个指向其关联事件的实例，而多个`std::shared_future`的实例则可以指向同一个事件。对后者而言，所有实例将同时变为**就绪**，并且它们都可以访问所有与该事件相关联的数据。这些关联的数据，就是这两种 future 成为模板的原因；像 `std::unique_ptr` 和 `std::shared_ptr` 一样，模板参数就是关联数据的类型。`std:future<void>`、`std::shared_future<void>`模板特化应该用于无关联数据的场合。虽然 future 被用于线程间通信，但是 future 对象本身却并不提供同步访问。如果多个线程需要访问同一个 future 对象，它们必须通过互斥元或其他同步机制来保护访问，如第 3 章中所述。然而，正如你将在 4.2.5 节中看到的，多个线程可以分别访问自己的 `std::shared_future<>`副本而无需进一步的同步，即使它们都指向同一个异步结果。

最基本的一次性事件是在后台运行着的计算结果。早在第 2 章中你就看到过 `std::thread` 并没有提供一种简单的从这一任务中返回值的方法，我保证这将在第 4 章中通过使用 future 加以解决——现在是时候去看看如何做了。

### 4.2.1　从后台任务中返回值

假设你有一个长期运行的计算，预期最终将得到一个有用的结果，但是现在，你还不需要这个值。也许你已经找到一种方法来确定生命、宇宙及万物的答案，这是从 Douglas Adams[1]那里偷来的一个例子。你可以启动一个新的线程来执行该计算，但这也意味着你必须注意将结果传回来，因为 `std::thread` 并没有提供直接的机制来这样做。这就是 `std::async` 函数模板（同样声明于`<future>`头文件中）的由来。

在不需要立刻得到结果的时候，你可以使用 `std::async` 来启动一个**异步任务**（**asynchronous task**）。`std::async` 返回一个 `std::future` 对象，而不是给你一个 `std::thread` 对象让你在上面等待，`std::future` 对象最终将持有函数的返回值。当你需要这个值时，只要在 future 上调用 `get()`，线程就会阻塞直到 future **就绪**，然后返回该值。清单 4.6 展示了一个简单的例子。

---

[1] 在 *The Hitchhiker's Guide to the Galaxy* 一书中，计算机 Deep Thought 被建造是用于决定“生命、宇宙和万物的答案”。答案是 42。

**清单 4.6　使用 std::future 获取异步任务的返回值**

```
#include <future>
#include <iostream>

int find_the_answer_to_ltuae();
void do_other_stuff();
int main()
{
    std::future<int> the_answer=std::async(find_the_answer_to_ltuae);
    do_other_stuff();
    std::cout<<"The answer is "<<the_answer.get()<<std::endl;
}
```

std::async 允许你通过将额外的参数添加到调用中，来将附加参数传递给函数，这与 std::thread 是同样的方式。如果第一个参数是指向成员函数的指针，第二个参数则提供了用来应用该成员函数的对象（直接地，或通过指针，或封装在 std::ref 中），其余的参数则作为参数传递给该成员函数。否则，第二个及后续的参数将作为参数，传递给第一个参数所指定的函数或可调用对象。和 std::thread 一样，如果参数是右值，则通过**移动**原来的参数来创建副本。这就允许使用只可移动的类型同时作为函数对象和参数。请看清单 4.7。

**清单 4.7　使用 std::async 来将参数传递给函数**

```
#include <string>
#include <future>
struct X
{
    void foo(int,std::string const&);
    std::string bar(std::string const&);
};
X x;
auto f1=std::async(&X::foo,&x,42,"hello");      // 调用 p->foo(42,"hello")，其中 p 是&x
auto f2=std::async(&X::bar,x,"goodbye");        // 调用 tmpx.bar("goodbye")，其中 tmpx 是 x 的副本
struct Y
{
    double operator()(double);
};
Y y;
auto f3=std::async(Y(),3.141);                  // 调用 tmpy(3.141)，其中 tmpy 是从 Y()移动构造的
auto f4=std::async(std::ref(y),2.718);          // 调用 y(2.718)
X baz(X&);
std::async(baz,std::ref(x));                    // 调用 baz(x)
class move_only
{
public:
    move_only();
    move_only(move_only&&)
    move_only(move_only const&) = delete;
    move_only& operator=(move_only&&);
    move_only& operator=(move_only const&) = delete;
```

```
    void operator()();
};
auto f5=std::async(move_only());
```
调用 tmp()，其中 tmp 是从 std::move(move_only()) 构造的

默认情况下，std::async 是否启动一个新线程，或者在等待 future 时任务是否同步运行都取决于具体实现方式。在大多数情况下这就是你所想要的，但你可以在函数调用之前使用一个额外的参数来指定究竟使用何种方式。这个参数为 std::launch 类型，可以是std::launch::deferred，以表明该函数调用将会延迟，直到在 future 上调用 wait() 或 get()为止，或者是 std::launch::async，以表明该函数必须运行在它自己的线程上，又或者是 std::launch::deferred | std::launch::async，以表明可以由具体实现来选择。最后一个选项是默认的。如果函数调用被延迟，它有可能永远都不会实际运行。例如，

```
auto f6=std::async(std::launch::async,Y(),1.2);           // 在新线程中运行
auto f7=std::async(std::launch::deferred,baz,std::ref(x)); // 在 wait()或 get()中运行
auto f8=std::async(
   std::launch::deferred | std::launch::async,
   baz,std::ref(x));                                       // 由具体实现来选择
auto f9=std::async(baz,std::ref(x));                       // 由具体实现来选择
f7.wait();                                                 // 调用延迟了的函数
```

正如你稍后将在本章看到，并将在第 8 章中再次看到的那样，使用 std::async 能够轻易地将算法转化成可以并行运行的任务。然而，这并不是将 std::future 与任务相关联的唯一方式；你还可以通过将任务封装在 std::packaged_task<>类模板的一个实例中，或者通过编写代码，用 std::promise<>类模板显式设置值等方式来实现。std::packaged_task 是比 std::promise 更高层次的抽象，所以我将从它开始。

### 4.2.2 将任务与 future 相关联

std::packaged_task<>将一个 future 绑定到一个函数或可调用对象上。当 std::packaged_task<>对象被调用时，它就调用相关联的函数或可调用对象，并且让 future 就绪，将返回值作为关联数据储存。这可以被用作线程池的构件（参见第 9 章），或者其他任务管理模式，例如在每个任务自己的线程上运行，或在一个特定的后台线程上按顺序运行所有任务。如果一个大型操作可以分成许多自包含的子任务，其中每一个都可以被封装在一个 std::packaged_task<>实例中，然后将该实例传给任务调度器或线程池。这样就抽象出了任务的详细信息，调度程序仅需处理 std::packaged_task<>实例，而非各个函数。

std::packaged_task<>类模板的模板参数为函数签名，比如 void()表示无参数无返回值的函数，或是像 int(std::string&,double*)表示接受对 std::string 的非 const 引用和指向 double 的指针，并返回 int 的函数。当你构造

std::packaged_task 实例的时候，你必须传入一个函数或可调用对象，它可以接受指定的参数并且返回指定的返回类型。类型无需严格匹配，你可以用一个接受 int 并返回 float 的函数构造 std::packaged_task<double(double)>，因为这些类型是可以隐式转换的。

指定的函数签名的返回类型确定了从 get_future() 成员函数返回的 std::future<>的类型，而函数签名的参数列表用来指定封装任务的函数调用运算符的签名。例如，std::packaged_task<std::string(std::vector<char>*,int)>的部分类定义如清单 4.8 所示。

**清单 4.8 std::packaged_task<>特化的部分类定义**

```
template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
    template<typename Callable>
    explicit packaged_task(Callable&& f);
    std::future<std::string> get_future();
    void operator()(std::vector<char>*,int);
};
```

该 std::packaged_task 对象是一个可调用对象，它可以被封装入一个 std::function 对象，作为线程函数传给 std::thread，或传给需要可调用对象的另一个函数，或者干脆直接调用。当 std::packaged_task 作为函数对象被调用时，提供给函数调用运算符的参数被传给所包含的函数，并且将返回值作为异步结果，存储在由 get_future() 获取的 std::future 中。因此，你可以将任务封装在 std::packaged_task 中，并且在把 std::packaged_task 对象传到别的地方进行适当调用之前获取 future。当你需要结果时，你可以等待 future 变为就绪，清单 4.9 的例子实际展示了这一点。

### 在线程之间传递任务

许多 GUI 框架要求从特定的线程来完成 GUI 的更新，所以，如果另一个线程需要更新 GUI，它必须向正确的线程发送消息来实现这一点。std::packaged_task 提供了一种更新 GUI 的方法，该方法无需为每个与 GUI 相关的活动获取自定义的消息。

**清单 4.9 使用 std::packaged_task 在 GUI 线程上运行代码**

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
```

```
#include <utility>

std::mutex m;
std::deque<std::packaged_task<void()> > tasks;

bool gui_shutdown_message_received();
void get_and_process_gui_message();

void gui_thread()    ❶
{
    while(!gui_shutdown_message_received())    ❷
    {
        get_and_process_gui_message();    ❸
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lk(m);
            if(tasks.empty())    ❹
                continue;
            task=std::move(tasks.front());    ❺
            tasks.pop_front();
        }
        task();    ❻
    }
}

std::thread gui_bg_thread(gui_thread);
template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
    std::packaged_task<void()> task(f);    ❼
    std::future<void> res=task.get_future();    ❽
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));    ❾
    return res;    ❿
}
```

此代码非常简单：GUI 线程 ❶ 循环直到收到通知 GUI 停止的消息 ❷，反复轮询待处理的 GUI 消息 ❸，比如用户点击，以及任务队列中的任务。如果队列中没有任务 ❹，则再次循环；否则，从任务队列中提取任务 ❺，解除队列中的锁，并运行任务 ❻。当任务完成时，与该任务相关联的 future 被设为就绪。

在队列上发布任务也同样简单：利用所提供的函数创建一个新的任务包❼，通过调用 `get_future()`成员函数从任务中获取 future❽，同时在返回 future 到调用处之前❾将任务置于列表之上❿。发出消息给 GUI 线程的代码如果需要知道任务已完成，则可以等待该 future；若无需知道，则可以丢弃该 future。

本示例中的任务使用 `std:packaged_task<void()>`，它封装了一个接受零参数且返回 `void` 的函数或可调用对象（如果它返回了别的东西，则返回值被丢弃）。这是最简单的任务，但如你在前面所看到的，`std:packaged_task` 也可以用于更复杂的

情况——通过指定一个不同的函数签名作为模板参数，你可以改变返回类型（以及在 future 的关联状态中存储的数据类型）和函数调用运算符的参数类型。这个示例可以进行简单的扩展，让那些在 GUI 线程上运行的任务接受参数，并且返回 std::future 中的值，而不是仅一个完成指示符。

那些无法用一个简单函数调用表达的任务和那些结果可能来自不止一个地方的任务又当如何？这些情况都可以通过第三种创建 future 的方式来处理：使用 std::promise 来显式地设置值。

## 4.2.3 生成(std::)promise

当有一个需要处理大量网络连接的应用程序时，通常倾向于在独立的线程上分别处理每个连接，因为这能使得网络通信更易于理解也更易于编程。这对于较低的连接数（因而线程数也较低）效果很好。然而，随着连接数的增加，这就变得不那么适合了；大量的线程就会消耗大量操作系统资源，并可能导致大量的上下文切换（当线程数超过了可用的硬件并发），进而影响性能。在极端情况下，操作系统可能会在其网络连接能力用尽之前，就为运行新的线程而耗尽资源。在具有超大量网络连接的应用程序中，通常用少量线程（可能仅有一个）来处理连接，每个线程一次处理多个连接。

考虑其中一个处理这种连接的线程。数据包将以基本上随机的顺序来自于待处理的各个连接，同样地，数据包将以随机顺序进行排队发送。在多数情况下，应用程序的其他部分将通过特定的网络连接，等待着数据被成功地发送或是新一批数据被成功地接收。

std::promise<T>提供一种设置值（类型 T）方式，它可以在这之后通过相关联的 std::future<T>对象进行读取。一对 std::promise/std::future 为这一设施提供了一个可能的机制；等待中的线程可以阻塞 future，同时提供数据的线程可以使用配对中的 promise 项，来设置相关的值并使 future 就绪。

你可以通过调用 get_future()成员函数来获取与给定的 std::promise 相关的 std::future 对象，比如 std::packaged_task。当设置完 promise 的值（使用 set_value()成员函数），future 会变为就绪，并且可以用来获取所存储的数值。如果销毁 std::promise 时未设置值，则将存入一个异常。4.2.4 节描述了异常是如何跨线程转移的。

清单 4.10 展示了处理如前文所述的连接的示例代码。在这个例子中，使用一对 std::promise<bool>/std::future<bool>对来标识一块传出数据的成功传输；与 future 关联的值就是一个简单的成功/失败标志。对于传入的数据包，与 future 关联的数据为数据包的负载。

清单 4.10 使用 promise 在单个线程中处理多个连接

```
#include <future>

void process_connections(connection_set& connections)
{
    while(!done(connections))    <-❶
    {                                    ❷
        for(connection_iterator    <-
                connection=connections.begin(),end=connections.end();
            connection!=end;
            ++connection)
        {
            if(connection->has_incoming_data())    <-❸
            {
                data_packet data=connection->incoming();
                std::promise<payload_type>& p=
                    connection->get_promise(data.id);    <-❹
                p.set_value(data.payload);
            }
            if(connection->has_outgoing_data())    <-❺
            {
                outgoing_packet data=
                    connection->top_of_outgoing_queue();
                connection->send(data.payload);
                data.promise.set_value(true);    <-❻
            }
        }
    }
}
```

函数 process_connections()一直循环到 done()返回 true❶。每次循环中，轮流检查每个连接❷，在有传入数据时获取之❸或是发送队列中的传出数据❺。此处假定一个输入数据包具有 ID 和包含实际数据在内的负载。此 ID 被映射至 std::promise（可能是通过在关联容器中进行查找）❹，并且该值被设为数据包的负载。对于传出的数据包，数据包取自传出队列，并实际上通过此连接发送。一旦发送完毕，与传出数据关联的 promise 被设为 true 以表示传输成功❻。此映射对于实际网络协议是否完好，取决于协议本身；这种 promise/future 风格的结构可能并不适用于某特定情况，尽管它确实与某些操作系统支持的异步 I/O 具有相似的结构。

迄今为止的所有代码完全忽略了异常。虽然想象一个万物都始终运转良好的世界是美好的，但却不切实际。有时候磁盘装满了，有时候你要找的东西恰好不在那里，有时候网络故障，有时数据库损坏。如果你正在需要结果的线程中执行操作，代码可能只是用异常报告了一个错误，因此仅仅因为你想用 std::packaged_task 或 std::promise，就限制性地要求所有事情都正常工作是不必要的。因此 C++标准库提供一个简便的方式，来处理这种场景下的异常，并允许他们作为相关结果的一部分而保存。

## 4.2.4 为 future 保存异常

考率下面简短的代码片段。如果将 -1 传入 square_root() 函数，会引发一个异常，同时将被调用者所看到。

```
double square_root(double x)
{
    if(x<0)
    {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}
```

现在假设不是仅从当前线程调用 square_root()：

```
double y=square_root(-1);
```

而是以异步调用的形式运行调用：

```
std::future<double> f=std::async(square_root,-1);
double y=f.get();
```

两者行为完全一致自然是最理想的；与 y 得到函数调用的任意一种结果一样，如果调用 f.get() 的线程能像在单线程情况下一样，能够看到里面的异常，那是极好的。

实际情况则是，如果作为 std::async 一部分的函数调用引发了异常，该异常会被储存在 future 中，代替所存储的值，future 变为**就绪**，并且对 get() 的调用会重新引发所存储的异常（注：重新引发的是原始异常对象抑或其副本，C++标准并没有指定，不同的编译器和库在此问题上作出了不同的选择）。这同样发生在将函数封装入 std::packaged_task 的时候——当任务被调用时，如果封装的函数引发异常，该异常代替结果存入 future，准备在调用 get() 时引发。

顺理成章，std::promise 在显式地函数调用下提供相同的功能。如果期望存储一个异常而不是一个值，则调用 set_exception() 成员函数而不是 set_value()。这通常是在引发异常作为算法的一部分时用在 catch 块中，将该异常填入 promise。

```
extern std::promise<double> some_promise;

try
{
    some_promise.set_value(calculate_value());
}
catch(...)
{
    some_promise.set_exception(std::current_exception());
}
```

这里使用 std::current_exception()来获取已引发的异常。作为替代，可以使用 std::copy_exception()直接存储新的异常而不对引发。

```
some_promise.set_exception(std::copy_exception(std::logic_error("foo ")));
```

在异常的类型已知时，这比使用 try/catch 块更为简洁，并且应该优先使用，这不仅仅简化了代码，也为编译器提供更多的优化代码的机会。

另一种将异常存储至 future 的方式，是销毁与 future 关联的 std::promise 或 std::packaged_task，而无需在 promise 上调用设置函数或是调用打包任务。在任何一种情况下，如果 future 尚未**就绪**，std::promise 或 std::packaged_task 的析构函数会将具有 std::future_errc::broken_promise 错误代码的 std::future_error 异常存储在相关联的状态中。通过创建 future，你承诺提供一个值或异常，而通过销毁该值或异常的来源，你违背了承诺。在这种情况下如果编译器没有将任何东西存进 future，等待中的线程可能会永远等下去。

到目前为止，所有的例子都使用了 std::future。然而，std::future 有其局限性，最起码，只有一个线程能等待结果。如果需要多于一个的线程等待同一个事件，则需要使用 std::shared_future 来代替。

### 4.2.5 等待自多个线程

尽管 std::future 能处理从一个线程向另一线程转移数据所需的全部必要的同步，但是调用某个特定的 std::future 实例的成员函数却并没有相互同步。如果从多个线程访问单个 std::future 对象而不进行额外的同步，就会出现**数据竞争**和未定义行为。这是有意为之的，std::future 模型统一了异步结果的所有权，同时 get()的单发性质使得这样的并发访问没有意义——只有一个线程可以获取值，因为在首次调用 get()后，就没有任何可获取的值留下了。

如果你的并发代码的绝妙设计要求多个线程能够等待同一个事件，目前还无需失去信心；std::shared_future 完全能够实现这一点。鉴于 std::future 是仅**可移动的**，所以所有权可以在实例间转移，但是一次只有一个实例指向特定的异步结果。std::shared_future 实例是**可复制的**，因此可以有多个对象引用同一个相关状态。

现在，即便有了 std::shared_future，各个对象的成员函数仍然是不同步的，所以为了避免从多个线程访问单个对象时出现数据竞争，必须使用锁来保护访问。首选的使用方式，是用一个对象的副本来代替，并且让每个线程访问自己的副本。从多个线程访问共享的异步状态，如果每个线程都是通过自己的 std::shared_future 对象去访问该状态，那么就是安全的，见图 4.1。

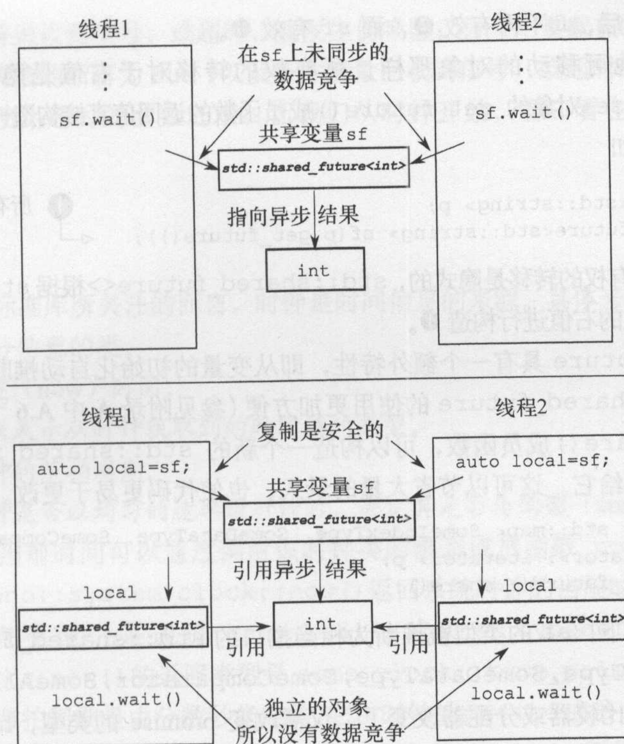



图 4.1 使用多个 std::shared_future 对象避免数据竞争

std::shared_future 的一个潜在用处，是实现类似复杂电子表格的并行执行。每个单元都有一个单独的终值，可以被多个其他单元格中的公式使用。用来计算各个单元格结果的公式可以使用 std::shared_future 来引用第一个单元。如果所有独立单元格的公式被并行执行，那些可以继续完成的任务可以进行，而那些依赖于其他单元格的公式将阻塞，直到其依赖关系准备就绪。这就使得系统能最大限度地利用可用的硬件并发。

引用了异步状态的 std::shared_future 实例可以通过引用这些状态的 std::future 实例来构造。由于 std::future 对象不和其他对象共享异步状态的所有权，因此该所有权必须通过 std::move 转移到 std::shared_future，将 std::future 留在空状态，就像它被默认构造了一样。

```
std::promise<int> p;
std::future<int> f(p.get_future());
assert(f.valid());                              ❶ future f 是有效的
std::shared_future<int> sf(std::move(f));
assert(!f.valid());                             ❷ f 不再有效
assert(sf.valid());   ❸ sf 现在有效
```

此处，future f 刚开始是有效的 ❶，因为它引用了 promise p 的异步状态，但是将状

态转移至 sf 后，f 不再有效 ❷，而 sf 有效 ❸。

正如其他可移动的对象那样，所有权的转移对于右值是隐式的，因此可以从 std::promise 对象的 get_future() 成员函数的返回值直接构造一个 std::shared_future，例如

```
std::promise<std::string> p;                          ❶ 所有权的隐式转移
std::shared_future<std::string> sf(p.get_future());  <-┘
```

此处，所有权的转移是隐式的，std::shared_future<>根据 std::future<std::string>类型的右值进行构造 ❶。

std::future 具有一个额外特性，即从变量的初始化自动推断变量类型的功能，使得 std::shared_future 的使用更加方便（参见附录 A 中 A.6 节）。std::future 具有一个 share() 成员函数，可以构造一个新的 std::shared_future，并且直接将所有权转移给它。这可以节省大量的录入，也使代码更易于更改。

```
std::promise< std::map< SomeIndexType, SomeDataType, SomeComparator,
    SomeAllocator>::iterator> p;
auto sf=p.get_future().share();
```

这种情况下，sf 的类型被推断为相当拗口的 std::shared_future<std::map<SomeIndexType,SomeDataType,SomeComparator,SomeAllocator>::iterator>。如果比较器或分配器改变了，仅需改变 promise 的类型，future 的类型将自动更新以匹配。

有些时候，你会想要限制等待事件的时长，无论是因为某段代码能够占用的时间有着硬性的限制，还是因为如果事件不会很快发生，线程就可以去做其他有用的工作。为了处理这个功能，许多等待函数具有能够指定超时的变量。

## 4.3 有时间限制的等待

前面介绍的所有阻塞调用都会阻塞一个不确定的时间段，挂起线程直至等待的事件发生。在许多情况下这是没问题的，但在某些情况下你会希望给等待时间加一个限制。这就使得能够发送某种形式的“我还活着”的消息给交互用户或者另一个进程，或是在用户已经放弃等待并且按下取消键时，干脆放弃等待。

有两类可供指定的超时：一为基于时间段的超时，即等待一个指定的时间长度（例如 30ms），或是绝对超时，即等到一个指定的时间点（例如世界标准时间 2011 年 11 月 30 日 17:30:15.045987023）。大多数等待函数提供处理这两种形式超时的变量，处理基于时间段超时的变量具有_for 后缀，而处理绝对超时的变量具有_until 后缀。

例如，std::condition_variable 具有两个重载版本的 wait_for()成员函数和两个重载版本的 wait_until()成员函数，对应于两个重载版本的 wait()——一个

重载只是等待到收到信号，或超时，或发生伪唤醒；另一个重载在唤醒时检测所给的断言，并只在所给的断言为 true（以及条件变量已收到信号）或超时的情况下才返回。

在细看使用超时的函数之前，让我们从时钟开始，看一看在 C++中是如何指定时间的。

### 4.3.1 时钟

就 C++标准库所关注的而言，时钟是时间信息的来源。具体来说，时钟是提供以下四个不同部分信息的类。

- 现在（**now**）时间。
- 用来表示从时钟获取到的时间值的类型。
- 时钟的节拍周期。
- 时钟是否以均匀的速率进行计时，决定其是否为匀速（**steady**）时钟。

时钟的当前时间可以通过调用该时钟类的静态成员函数 `now()`来获取。例如，`std::chrono::system_clock::now()`返回系统时钟的当前时间。对于具体某个时钟的时间点类型，是通过 `time_point` 成员的 `typedef` 来指定的，因此 `some_clock::now()`的返回类型是 `some_clock::time_point`。

时钟的节拍周期是由分数秒指定的，它由时钟的`period`成员`typedef`给出——每秒走 25 拍的时钟，就具有 `std::ratio<1,25>`的 `period`，而每 2.5 秒走一拍的时钟则具有 `std::ratio<5,2>`的 `period`。如果时钟的节拍周期直到运行时方可知晓，或者可能所给的应用程序运行期间可变，则 `period` 可以指定为平均的节拍周期、最小可能的节拍周期，或是编写类库的人认为合适的一个值。在所给的一次程序的执行中，无法保证观察到的节拍周期与该时钟所指定的 `period` 相符。

如果一个时钟以均匀速率计时（无论该时速是否匹配 `period`）且不能被调整，则该时钟被称为匀速（**steady**）时钟。如果时钟是匀速的，则时钟类的 `is_steady` 静态数据成员为 `true`，反之为 `false`。通常情况下，`std::chrono::system_clock` 是不匀速的，因为时钟可以调整，考虑到本地时钟漂移，这种调整甚至是自动执行的。这样的调整可能会引起调用 `now()`所返回的值，比之前调用 `now()`所返回的值更早，这违背了均匀计时速率的要求。如你马上要看到的那样，匀速时钟对于计算超时来说非常重要，因此 C++标准库提供形式为 `std::chrono::steady_clock` 的匀速时钟。由 C++标准库提供的其他时钟包括 `std::chrono::system_clock`（上文已提到），它代表系统的“真实时间”时钟，并为时间点和 `time_t` 值之间的相互转换提供函数，还有 `std::chrono::high_resolution_clock`，它提供所有类库时钟中最小可能的节拍周期（和可能的最高精度）。它实际上可能是其他时钟之一的 `typedef`。这些时钟与其他时间工具都定义在`<chrono>`类库头文件中。

我们马上要看看如何表示时间点，但在此之前，先来看看时间段是如何表示的。

## 4.3.2 时间段

时间段是时间支持中的最简单部分，它们是由std::chrono::duration<>类模板（线程库使用的所有C++时间处理工具均位于std::chrono的命名空间中）进行处理的。第一个模板参数为所代表类型（如int、long、或double），第二个参数是个分数，指定每个时间段单位表示多少秒。例如，以 short 类型存储的几分钟的数目表示为std::chrono::duration<short,std::ratio<60,1>>，因为1分钟有60秒。另一方面，以 double 类型存储的毫秒数则表示为 std::chrono::duration<double,std::ratio<1,1000>>，因为1毫秒为1/1000秒。

标准库在std::chrono命名空间中为各种时间段提供了一组预定义的typedef：nanoseconds、microseconds、milliseconds、seconds、minutes和hours。它们均使用一个足够大的整数类型以供表示，以至于如果你希望的话，可以使用合适的单位来表示超过500年的时间段。还有针对所有国际单位比例的typedef可供指定自定义时间段时使用，从std::atto（$10^{-18}$）至std::exa（$10^{18}$）（还有更大的，若你的平台具有 128 位整型类型），例如 std::duration<double,std::centi>是以double类型表示的1/100秒的计时。

在无需截断值的场合，时间段之间的转换是隐式的（因此将小时转换成秒是可以的，但将秒转换成小时则不然）。显式转换可以通过 std::chrono::duration_cast<>实现：

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
    std::chrono::duration_cast<std::chrono::seconds>(ms);
```

结果是截断而非四舍五入，因此在此例中s值为54。

时间段支持算术运算，因此可以加、减时间段来得到新的时间段，或者可以乘、除一个底层表示类型（第一个模板参数）的常数。因此5*seconds(1)和seconds(5)或minutes(1)-seconds(55)是相同的。时间段中单位数量的计数可以通过count()成员函数获取。因此std::chrono::milliseconds(1234).count()为1234。

基于时间段的等待是通过std::chrono::duration<>实例完成的。例如，可以等待future就绪最多35毫秒。

```
std::future<int> f=std::async(some_task);
if(f.wait_for(std::chrono::milliseconds(35))==std::future_status::ready)
    do_something_with(f.get());
```

等待函数都会返回一个状态以表示等待是否超时，或者所等待的事件是否发生。在这种情况下，你在等待一个future，若等待超时，函数返回std::future_status::

timeout，若 future 就绪，则返回 std::future_status::ready，或者如果 future 任务推迟，则返回 std::future_status::deferred。基于时间段的等待使用类库内部的匀速时钟来衡量时间，因此 35 毫秒意味着 35 毫秒的逝去时间，即便系统时钟在等待期间进行了调整（向前或向后）。当然，系统调度的多变和 OS 时钟的不同精度意味着线程之间发出调用并返回的实际时间可能远远长于 35 毫秒。

在看过时间段后，接下可以继续看看时间点。

### 4.3.3 时间点

时钟的时间点是通过 std::chrono::time_point<>类模板的实例来表示的，它以第一个模板参数指定其参考的时钟，并且以第二个模板参数指定计量单位（std::chrono::duration<>的特化）。时间点的值是时间的长度（指定时间段的倍数），因而一个特定时间点被称为时钟的纪元（**epoch**）。时钟的纪元是一项基本参数，但却不能直接查询，也未被 C++标准指定。典型的纪元包括 1970 年 1 月 1 日 00:00，以及运行应用程序的计算机引导启动的瞬间。时钟可以共享纪元或拥有独立的纪元。如果两个时钟共享一个纪元，则在一个类中的 time_point typedef 可指定另一个类作为与 time_point 相关联的时钟类型。虽然无法找出纪元的时间所在，但可以获取给定 time_point 的 time_since_epoch()。该成员函数返回一个时间段的值，其指定了从时钟纪元到该时间点的时间长度。

例如，你可以指定一个时间点为 std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>。这将保持时间与系统时钟相关，但却以分钟而不是系统时钟的固有测量精度（通常为秒或更小）进行测量。

你可以从 std::chrono::time_point<>的实例加上和减去时间段来产生新的时间点，因此 std::chrono::high_resolution_clock::now()+std::chrono::nanoseconds(500) 将在 future 中给你 500 纳秒的时间。这对于在知道代码块的最大时间段情况下计算绝对超时是极好的，但若其中有对等待函数的多个调用，或是在等待函数之前有非等待函数，就会占据一部分时间预算。

你还可以从另一个共享同一个时钟的时间点减去一个时间点。结果为指定两个时间点之间长度的时间段。这对于代码块的计时非常有用，例如，

```
auto start=std::chrono::high_resolution_clock::now();
do_something();
auto stop=std::chrono::high_resolution_clock::now();
std::cout<<"do_something() took "
  <<std::chrono::duration<double,std::chrono::seconds>(stop-start).count()
  <<" seconds"<<std::endl;
```

然而 std::chrono::time_point<>实例的时钟参数能做的不仅仅是指定纪元。当你将时间点传给到接受绝对超时的等待函数时，时间点的时钟参数可以用来测量时

间。当时钟改变时会产生一个重要的影响，因为这一等待会跟踪时钟的改变，并且在时钟的now()函数返回一个晚于指定超时的值之前都不会返回。如果时钟向前调整，将减少等待的总长度（按照匀速时钟计量），反之如果向后调整，就可能增加等待的总长度。

如你所料，时间点和等待函数的_until变种共同使用。典型用例是在用作从程序中一个固定点的某个时钟::now()开始的偏移量，尽管与系统时钟相关联的时间点可以通过在对用户可见的时间，用std::chrono::system_clock::to_time_point()静态成员函数从time_t转换而来。例如，如果有一个最大值为500毫秒的时间，来等待一个与条件变量相关的事件，你可以按清单4.11所示来做。

**清单4.11 等待一个具有超时的条件变量**

```
#include <condition_variable>
#include <mutex>
#include <chrono>
std::condition_variable cv;
bool done;
std::mutex m;
bool wait_loop()
{
    auto const timeout= std::chrono::steady_clock::now()+
        std::chrono::milliseconds(500);
    std::unique_lock<std::mutex> lk(m);
    while(!done)
    {
        if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
            break;
    }
    return done;
}
```

如果没有向等待传递断言，那么这是在有时间限制下等待条件变量的推荐方式。这种方式下，循环的总长度是有限的。如4.1.1节所示，当不传入断言时，需要通过循环来使用条件变量，以便处理伪唤醒。如果在循环中使用wait_for()，可能在伪唤醒前，就已结束等待几乎全部时长，并且在经过下一次等待开始后再来一次。这可能会重复任意次，使得总的等待时间无穷无尽。

在看过了指定超时的基础知识后，让我们来看看能够使用超时的函数。

### 4.3.4 接受超时的函数

超时的最简单用法，是将延迟添加到特定线程的处理过程中，以便在它无所事事的时候避免占用其他线程的处理时间。在4.1节你曾见过这样的例子，在循环中轮询一个“完成”标记。处理它的两个函数是 std::this_thread::sleep_for()和std::this_thread::sleep_until()。它们像一个基本的闹钟一样工作：在指定

时间段（使用sleep_for()）或直至指定的时间点（使用sleep_until()），线程进入睡眠状态。sleep_for()对于那些类似于4.1节中的例子是有意义的，其中一些事情必须周期性地进行，并且逝去的时间是重要的。另一方面，sleep_until()允许安排线程在特定时间点唤醒。这可以用来触发半夜里的备份，或在早上6:00打印工资条，或在做视频回放时暂停线程直至下一帧的刷新。

当然，睡眠并不是唯一的接受超时的工具。你已经看到了可以将超时与条件变量和future一起使用。如果互斥元支持的话，甚至可以试图在互斥元获得锁时使用超时。普通的std::mutex和std::recursive_mutex并不支持锁定上的超时，但是std::timed_mutex和std::recursive_timed_mutex支持。这两种类型均支持try_lock_for()和try_lock_until()成员函数，它们可以在指定时间段内或在指定时间点之前尝试获取锁。表4.1展示了C++标准库中可以接受超时的函数及其参数和返回值。列作时间段（**duration**）的参数必须为std::duration<>的实例，而那些列作time_point的必须为std::time_point<>的实例。

表4.1 接受超时的函数

| 类/命名空间 | 函数 | 返回值 |
|---|---|---|
| std::this_thread命名空间 | sleep_for(*duration*)<br>sleep_until(*time_point*) | 不可用 |
| std::condition_variable或<br>std::condition_variable_any | wait_for(*lock*, *duration*)<br>wait_until(*lock*, *time_point*) | std::cv_status::timeout或<br>std::cv_status::no_timeout |
| | wait_for(*lock*, *duration*, *predicate*)<br>wait_until(*lock*, *time_point*, *predicate*) | bool—当唤醒时**predicate**的返回值 |
| std::timed_mutex或std::recursive_timed_mutex | try_lock_for(*duration*)<br>try_lock_until(*time_point*) | bool—true如果获得了锁，否则false |
| std::unique_lock<TimedLockable> | unique_lock(*lockable*, *duration*)<br>unique_lock(*lockable*, *time_point*) | 不可用—owns_lock()在新构造的对象上；如果获得了锁返回true，否则false |
| | try_lock_for(*duration*)<br>try_lock_until(*time_point*) | bool—true如果获得了锁，否则false |
| std::future<*ValueType*>或std::shared_future<*ValueType*> | wait_for(*duration*)<br>wait_until(*time_point*) | std::future_status::timeout如果等待超时，<br>std::future_status::ready如果future就绪或<br>std::future_status::deferred如果future持有的延迟函数还没有开始 |

目前，我已经介绍了条件变量、future、promise 和打包任务的机制，接下来是时候看一看更广的图景，以及如何利用它们来简化线程间操作的同步。

## 4.4 使用操作同步来简化代码

使用截至目前在本章中描述的同步工具作为构建模块，允许你着重关注需要同步的操作而非机制。一种可以简化代码的方式，是采用一种更加函数式的（**functional**，在函数式编程意义上）的方法来编写并发程序。并非直接在线程之间共享数据，而是每个任务都可以提供它所需要的数据，并通过使用 future 将结果传播至需要它的线程。

### 4.4.1 带有 future 的函数式编程

函数式编程（**functional programming, FP**）指的是一种编程风格，函数调用的结果仅单纯依赖于该函数的参数而不依赖于任何外部状态。这与函数的数学概念相关，同时也意味着如果用同一个参数执行一个函数两次，结果是完全一样的。这是许多 C++标准库中数学函数，如 `sin`、`cos` 和 `sqrt`，以及基本类型简单操作如 `3+3`、`6*9`、或 `1.3/4.7` 的特性。纯函数也不**修改**任何外部状态，函数的影响完全局限在返回值上。

这使得事情变得易于思考，尤其当涉及并发时，因为第 3 章中讨论的许多与共享内存相关的问题不复存在。如果没有修改共享数据，那么就不会有竞争条件，因此也就没有必要使用互斥元来保护共享数据。这是一个如此强大的简化，使得诸如 Haskell[1]这样的编程语言，在**默认情况**下其所有函数都是纯函数，开始在编写并发系统中变得更为流行。因为大多数东西都是纯的，实际上的**确**修改共享状态的非纯函数就显得鹤立鸡群，因而也更易于理解它们是如何纳入应用程序整体结构的。

然而函数式编程的好处不仅仅局限在那些将其作为默认范型的语言。C++是一种多范型语言，它完全可以用 FP 风格编写程序。随着 lambda 函数（参见附录 A 中 A.6 节）的到来，从 Boost 到 TR1 的 `std::bind` 合并，和自动变量类型推断的引入（参见附录 A 中 A.7 节），C++11 比 C++98 更为容易实现函数式编程。future 是使得 C++中 FP 风格的并发切实可行的最后一块拼图。一个 future 可以在线程间来回传递，使得一个线程的计算结果依赖于另一个的结果，而无需任何对共享数据的显式访问。

#### 1. FP 风格快速排序

为了说明在 FP 风格并发中 future 的使用，让我们来看一个简单的快速排序算法的实现。算法的基本思想很简单，给定一列值，取一个元素作为中轴，然后将列表分为两

[1] 参见 http://www.haskell.org/。

组——比中轴小的为一组，大于等于中轴的为一组。列表的已排序副本，可以通过对这两组进行排序，并按照先是比中轴小的值已排序列表，接着是中轴，再后返回大于等于中轴的值已排序列表的顺序进行返回来获取。图 4.2 展示了 10 个整数的列表是如何根据此步骤进行排序的。FP 风格的顺序实现在随后的代码中展示，它通过值的形式接受并返回列表，而不是像 `std::sort()` 那样就地排序。

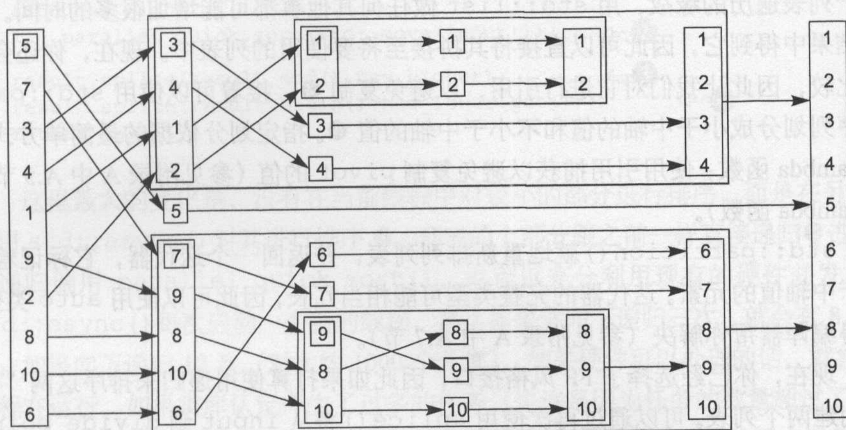



图 4.2　FP 风格递归排序

**清单 4.12　快速排序的顺序实现**

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());    ❶
    T const& pivot=*result.begin();                        ❷

    auto divide_point=std::partition(input.begin(),input.end(),
            [&](T const& t){return t<pivot;});             ❸

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin()
        divide_point);                                     ❹
    auto new_lower(
        sequential_quick_sort(std::move(lower_part)));     ❺
    auto new_higher(
        sequential_quick_sort(std::move(input)));          ❻

    result.splice(result.end(),new_higher);                ❼
```

```
    result.splice(result.begin(),new_lower);                  <—❽
    return result;
}
```

尽管接口是 FP 风格的，如果你自始至终使用 FP 风格，就会制作很多的副本，即你在内部使用了“标准”祈使风格。取出第一个元素作为中轴，方法是用 splice() ❶ 将其从列表前端切下。虽然这样可能会导致一个次优排序（考虑到比较和交换的次数），由于列表遍历的缘故，用 std::list 做任何其他事都可能增加很多的时间。你已知要在结果中得到它，因此可以直接将其拼接至将要使用的列表中。现在，你还会想要将其作比较，因此让我们对它进行引用，以避免复制 ❷。接着可以使用 std::partition 将序列划分成小于中轴的值和不小于中轴的值 ❸。指定划分依据的最简单方式是使用一个 lambda 函数；使用引用捕获以避免复制 pivot 的值（参见附录 A 中 A.5 节更多地了解 lambda 函数）。

std::partition()就地重新排列列表，并返回一个迭代器，它标记着第一个不小于中轴值的元素。迭代器的完整类型可能相当冗长，因此可以使用 auto 类型说明符，使得编译器帮你解决（参见附录 A 中 A.7 节）。

现在，你已经选择了 FP 风格接口，因此如果打算使用递归来排序这两“半”，则需要创建两个列表。可以通过再次使用 splice()将从 input 到 divide_point 的值移动至一个新的列表：lower_part❹。这使得 input 中只仅留下剩余的值。你可以接着用递归调用对这两个列表进行排序 ❺、❻。通过使用 std::move()传入列表，也可以在此处避免复制——但是结果也将被隐式地移动出来。最后，你可以再次使用 splice() 将 result 以正确的顺序连起来。new_higher 值在中轴之后直到末尾❼，而 new_lower 值从开始起，直到中轴之前❽。

### 2. FP 风格并行快速排序

由于已经使用了函数式风格，通过 future 将其转换成并行版本就很容易了，如清单 4.13 所示。这组操作与之前相同，区别在于其中一部分现在并行地运行。此版本使用 future 和函数式风格实现快速排序算法。

**清单 4.13 使用 future 的并行快速排序**

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());
    T const& pivot=*result.begin();
```

```
    auto divide_point=std::partition(input.begin(),input.end(),
            [&](T const& t){return t<pivot;});
    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);
                                                        ❶
    std::future<std::list<T> > new_lower(            <--
        std::async(&parallel_quick_sort<T>,std::move(lower_part)));

    auto new_higher(
        parallel_quick_sort(std::move(input)));   <--❷

    result.splice(result.end(),new_higher);       <--❸
    result.splice(result.begin(),new_lower.get());    <--❹
    return result;
}
```

这里最大的变化是，没有在当前线程中对较小的部分进行排序，而是在另一个线程中使用 std::async()对其进行排序❶。列表的上部分跟之前一样直接递归❷进行排序。通过递归调用 parallel_quick_sort()，你可以充分利用现有的硬件并发能力。如果 std::async()每次启动一个新的线程，那么如果你向下递归三次，就会有 8 个线程在运行；如果向下递归 10 次（对大约 1000 个元素），如果硬件可以处理的话，就将有 1024 个线程在运行。如果类库认定产生了过多的任务（也许是因为任务的数量超过了可用的硬件并发能力），则可能改为同步地产生新任务。他们会在调用 get()的线程中运行，而不是在新的线程中，从而避免在不利于性能的情况下把任务传递到另一个线程的开销。值得注意的是，对于 std::async 的实现来说，只有显式指定了 std::launch::deferred 再为每一个任务开启一个新线程（甚至在面对大量的过度订阅时），或是只有显式指定了 std::launch::async 再同步运行所有任务，才是完全符合的。如果依靠类库进行自动判定，建议你查阅一下这一实现的文档，看一看它究竟表现出什么样的行为。

与其使用 std::async()，不如自行编写 spawn_task()函数作为 std::packaged_task 和 std::thread 的简单封装，如清单 4.14 所示，为函数调用结果创建了一个 std::packaged_task，从中获取 future，在线程中运行之并返回 future。其本身并不会带来多少优点（实际上可能导致大量的过度订阅），但它为迁移到一个更复杂的实现做好准备，它通过一个工作线程池，将任务添加到一个即将运行的队列里。我们会在第 9 章研究线程池。相比于使用 std::async，只有在你确实知道将要做什么，并且希望想要通过线程池建立的方式进行完全掌控和执行任务的时候，才值得首选这种方法。

总之，回到 parallel_quick_sort。因为只是使用直接递归获取 new_higher，你可以将其拼接到之前的位置❸。但是现在 new_lower 是 std::future<std::list<T>>而不仅是列表，因此需要在调用 splice()之前调用 get()来获取值❹。这就会等待后台任务完成，并将结果**移动**至 splice()调用；get()返回一个引用了所包含结果的右值，所以它可以被移除（参见附录 A 中 A.1.1 节更多地了解右值引用和移动语义）。

即使假设 std::async()对可用的硬件并发能力进行最优化的使用，这仍不是快

速排序的理想并行实现。原因之一是，std::partition 完成了很多工作，且仍为一个连续调用，但对于目前已经足够好了。如果你对最快可能的并行实现有兴趣，请查阅学术文献。

**清单 4.14 一个简单 spawn_task 的实现**

```
template<typename F,typename A>
std::future<std::result_of<F(A&&)>::type>
    spawn_task(F&& f,A&& a)
{
    typedef std::result_of<F(A&&)>::type result_type;
    std::packaged_task<result_type(A&&)>
        task(std::move(f)));
    std::future<result_type> res(task.get_future());
    std::thread t(std::move(task),std::move(a));
    t.detach();
    return res;
}
```

函数式编程并不是唯一的避开共享可变数据的并发编程范式；另一种范式为 CSP（Communicating Sequential Process，通信顺序处理）[1]，这种范式下线程在概念上完全独立，没有共享数据，但是具有允许消息在它们之间进行传递的通信通道。这是被编程语言 Erlang（http://www.erlang.org/）所采用的范式，也通常被 MPI（Message Passing Interface，消息传递接口）（http://www.mpi-forum.org/）环境用于 C 和 C++中的高性能计算。我可以肯定到目前为止，你不会对这在 C++中也可以在一些准则下得到支持而感到意外，接下来的一节将讨论实现这一点的一种方式。

### 4.4.2 具有消息传递的同步操作

CSP 的思想很简单：如果没有共享数据，则每一个线程可以完全独立地推理得到，只需基于它对所接收到的消息如何进行反应。因此每个线程实际上可以等效为一个状态机：当它接收到消息时，它会根据初始状态进行操作，并以某种方式更新其状态，并可能向其他线程发送一个或多个消息。编写这种线程的一种方式，是将其形式化并实现一个有限状态机模型，但这并不是唯一的方式。状态机在应用程序结构中可以是隐式的。在任一给定的情况下，究竟哪种方法更佳，取决于具体形势的行为需求和编程团队的专长。但是选择实现每一个线程，则将其分割成独立的进程可能会从共享数据并发中移除很多复杂性，因而使得编程更加容易，降低了错误率。

真正的通信序列进程并不共享数据，所有的通信都通过消息队列，但由于 C++线程共享一个地址空间，因此不可能强制执行这一需求。这就是准则介入的地方。作为应用

[1] *Communicating Sequential Processes*, C.A.R. Hoare, Prentice Hall, 1985. Available free online at http://www.usingcsp.com/cspbook.pdf

程序或类库的作者，我们的责任是确保我们不在线程间共享数据。当然，为了线程之间的通信，消息队列必须是共享的，但是其细节可以封装在类库内。

想象一下，你正在实现 ATM 的代码。此代码需要处理人试图取钱的交互和与相关银行的交互，还要控制物理机器接受此人的卡片，显示恰当的信息，处理按键，出钞并退回用户的卡片。

处理这一切事情的其中一种方法，是将代码分成三个独立的线程：一个处理物理机器、一个处理 ATM 逻辑、还有一个与银行进行通信。这些线程可以单纯地通过传递消息而非共享数据进行通信。例如，当有人在机器旁边将卡插入或按下按钮时，处理机器的线程可以发送消息至逻辑线程，同时逻辑线程将发送消息至机器线程，指示要发多少钱等等。

对 ATM 逻辑进行建模的一种方式，是将其视为一个状态机。在每种状态中，线程等待可接受的消息，然后对其进行处理。这样可能会转换到一个新的状态，并且循环继续。一个简单实现中所涉及的状态如图 4.3 所示。在这个简化了的实现中，系统等待卡片插入。一旦卡片插入，便等待用户输入其密码，每次一个数字。他们可以删除最后输入的数字。一旦输入的数字足够多，就验证密码。如果密码错误，则结束，将卡片退回给用户，并继续等待有人插入卡片。如果密码正确，则等待用户取消交易或选择取出的金额。如果用户取消，则结束，并退出银行卡。如果用户选择了一个金额，则等待银行确认后发放现金并退出其卡片，或者显示“资金不足”的消息并退出其卡片。显然，真正的 ATM 比这复杂得多，但这已经足以阐述整个想法。

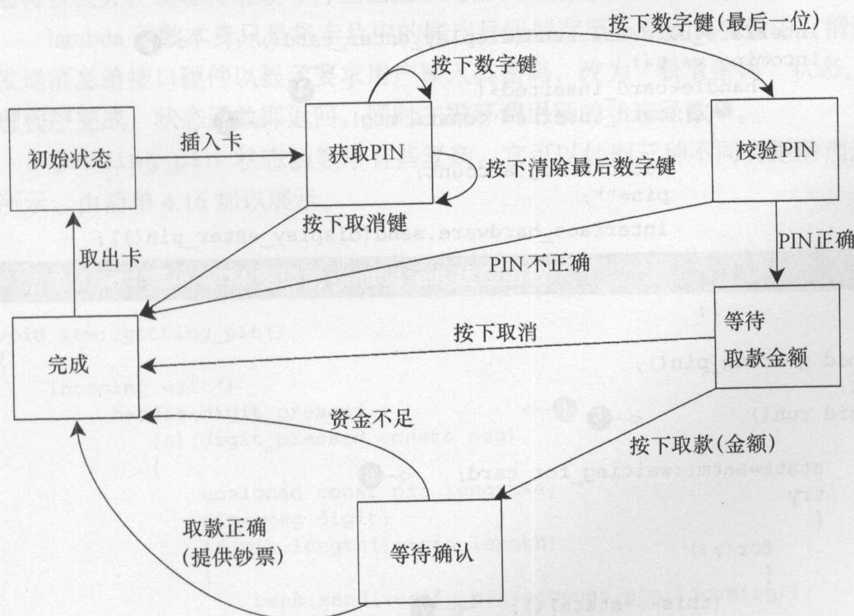



图 4.3 ATM 的简单状态机模型

有了 ATM 逻辑的状态机设计之后，就可以使用一个具有表示每一个状态的成员函数的类来实现之。每一个成员函数都可以等待指定的一组传入消息，并在它们到达后进行处理，其中可能触发切换到另一状态。每个不同的消息类型可以由一个独立的 struct 来表示。清单 4.15 展示了这样一个系统中 ATM 逻辑的部分简单实现，包括主循环和第一个状态的实现，即等待卡片插入。

如你所见，所有消息传递所必需的同步完全隐藏于消息传递库内部（其基本实现以及本示例的完整代码见附录 C）。

**清单 4.15 ATM 逻辑类的简单实现**

```
struct card_inserted
{
    std::string account;
};
class atm
{
    messaging::receiver incoming;
    messaging::sender bank;
    messaging::sender interface_hardware;
    void (atm::*state)();
    std::string account;
    std::string pin;

    void waiting_for_card()                                    <-❶
    {
        interface_hardware.send(display_enter_card());              <-❷
        incoming.wait()                                        <-❸
            .handle<card_inserted>(
                [&](card_inserted const& msg)                 <-❹
                {
                    account=msg.account;
                    pin="";
                    interface_hardware.send(display_enter_pin());
                    state=&atm::getting_pin;
                }
                );
    }
    void getting_pin();
public:
    void run()                       <-❺
    {
        state=&atm::waiting_for_card;         <-❻
        try
        {
            for(;;)
            {
                (this->*state)();        <-❼
            }
        }
```

```
            catch(messaging::close_queue const&)
            {
            }
        }
};
```

正如已经提到的，这里所描述的实现是从 ATM 所需的真正逻辑中粗略简化而来的，但这确实给你一种消息传递的编程风格的感受。没有必要去考虑同步和并发的问题，只要考虑在任意一个给定的店，可能会接收到哪些消息，以及该发送哪些消息。ATM 逻辑的状态机在单线程上运行，而系统的其他部分，比如银行的接口和终端的接口，则在独立的线程上运行。这种程序设计风格被称作行为**角色模型**（**actor model**）——系统中有多个离散的**角色**（均运行在独立线程上），用来相互发送消息以完成手头任务，除了直接通过消息传递的状态外，没有任何共享状态。

执行是由 run() 成员函数开始的 ❺，设置初始状态为 waiting_for_card❻ 并重复执行代表当前状态（不管它是什么）的成员函数❼。状态函数就是 atm 类的简单成员函数。waiting_for_card 状态函数 ❶ 也很简单：发送一则消息至界面以显示“等待卡片”的消息 ❷，接着等待要处理的消息。这里能处理的唯一消息类型是 card_inserted 消息 ❸，可以通过 lambda 函数进行处理 ❹。你可以将任意函数或函数对象传递给 handle 函数，但是像这种简单的情况，用 lambda 是最容易的。注意 handle() 函数调用是链接至 wait() 函数的。如果接收到的消息不匹配指定的类型，它将被丢弃，线程将继续等待直至接收到匹配的消息。

lambda 函数本身只是将卡片中的账户号码缓存至一个成员变量中，清除当前密码，发送消息给接口硬件以显示要求用户输入其密码，改为“获取密码”状态。一旦消息处理程序完成，状态函数即返回，同时主循环调用新的状态函数❼。

getting_pin 状态函数略有些复杂，它可以处理三种不同类型的消息，如图 4.3 所示。由清单 4.16 加以展示。

**清单 4.16　简单 ATM 实现的 getting_pin 状态函数**

```
void atm::getting_pin()
{
    incoming.wait()
        .handle<digit_pressed>(          <—❶
            [&](digit_pressed const& msg)
            {
                unsigned const pin_length=4;
                pin+=msg.digit;
                if(pin.length()==pin_length)
                {
                    bank.send(verify_pin(account,pin,incoming));
                    state=&atm::verifying_pin;
                }
```

```
        }
        )
    .handle<clear_last_pressed>(                    ❷
        [&](clear_last_pressed const& msg)
        {
            if(!pin.empty())
            {
                pin.resize(pin.length()-1);
            }
        }
        )
    .handle<cancel_pressed>(                        ❸
        [&](cancel_pressed const& msg)
        {
            state=&atm::done_processing;
        }
        );
}
```

这时，有三种能够处理的消息类型，所以 wait() 函数有三个 handle() 调用链接至结尾 ❶、❷、❸。每一次对 handle() 的调用将消息类型指定为模板参数，然后传至接受该特定消息类型作为参数的 lambda 函数。由于调用通过这种方式链接起来，wait() 的实现知道它正在等待一个 digit_pressed 消息、clear_last_pressed 消息或是 cancel_pressed 消息。任何其他类型的消息将再次被丢弃。

这时，在获取消息之后你并不一定非要改变状态。例如，如果你得到了一个 digit_pressed 消息，仅需将它添加至 pin，除非它为最后的数字。清单 4.15 中的主循环❼将再次调用 getting_pin() 来等待下一个数字（或是清除或取消）。

这对应于图 4.3 所示的行为。每一个状态框通过不同的成员函数来实现，它们等待相关的消息并适当地更新状态。

如你所见，这种编程风格可以大大简化设计并发系统的任务，因为每个线程可以完全独立地对待。这就是使用多线程来划分关注点的一个例子，并且需要你明确决定如何在线程间划分任务。

## 4.5 小结

线程间的同步操作，是编写一个使用并发的应用程序的重要组成部分。如果没有同步，那么线程本质上是独立的，就可以被写作独立的应用程序，由于它们之间存在相关活动而作为群组运行。在本章中，我介绍了同步操作的各种方式，从最基本条件变量，到 future、promise 以及打包任务。我还讨论了解决同步问题的方式，函数式编程，其中每个任务产生的结果完全依赖于它的输入而不是外部环境，以及消息传递，其中线程间的通信是通过一个扮演中介角色的消息子系统发送异步消息来实现的。

我们已经讨论了许多在 C++中可用的高阶工具，现在是时候来看看令这一切得以运转的底层工具了：C++内存模型和原子操作。

# 第 5 章　C++内存模型和原子类型上操作

本章主要内容

- C++11 内存模型的详情
- 由 C++标准库提供的原子类型
- 这些类型上可用的操作
- 如何使用那些操作提供线程间的同步

C++11 标准中最重要的特性之一，是大多数程序员甚至都不会关注的东西。它并不是新的语法特性，也不是新的类库功能，而是新的多线程感知内存模型。若是没有内存模型来严格定义基本构造模块如何工作，我所介绍的功能就不能可靠地工作。当然，大多数程序员没有注意到它是有原因的。如果使用互斥元来保护数据，以及条件变量或者 future 作为事件信号，它们为何工作的细节就不重要了。仅当你开始尝试“接近机器”时，内存模型的精确细节才重要。

无论其他语言如何，C++是一门系统编程语言。标准委员会的目标之一，是不再需要一个比 C++更低级的语言。应该在 C++里为程序员提供足够的灵活性，在做任何他们需要的事情时不会被语言挡在路中间，并且在提出需求时允许他们“接近机器”。原子类型和操作正是要允许这一点，提供了可通常减至一或两个 CPU 指令的低阶同步操作的功能。

在本章中，我将从阐述内存模型的基础开始，接着转到原子类型和操作，并最终介绍可用于原子类型上操作的多种同步类型。这是相当复杂的，除非你打算使用原子操作

编写代码以实现同步（比如第7章中的无锁数据结构），否则无需知道这些细节。

让我们放轻松，来看看内存模型的基础。

## 5.1 内存模型基础

内存模型包括两个方面：基本**结构**方面，这与事物是如何放置在内存中的有关；然后是**并发**方面。结构方面对于并发是很重要的，尤其从低级原子操作中来看，因此我将从这里开始。在C++中，一切都是关于对象和内存位置。

### 5.1.1 对象和内存位置

C++程序中的所有数据均是由**对象（object）**组成的。这并不是说你可以创建一个派生自 int 的新类，或是基本类型具有成员函数，或者当人们谈及如 Smalltalk 或 Ruby 这样的语言，说“一切都是对象”时暗指的任何其他结果。这只是一句关于C++中数据的构造块的一种陈述。C++标准定义对象为“存储区域”，尽管它会为这些对象分配属性，如它们的类型和生存期。

其中一些对象是简单基本类型的值，如 int 或 float，而另一些则是用户定义类的实例。有些对象（例如数组、派生类的实例和具有非静态数据成员类的实例）具有子对象，但其他的则没有。

无论什么类型，对象均被存储于一个或多个**内存位置**中。每个这样的内存位置要么是一个标量类型的对象（或子对象），比如 unsigned short 或 my_class*，要么是相邻位域的序列。如果使用位域，有非常重要的一点必须注意：虽然相邻的位域是不同的对象，但它们仍然算作相同的内存位置。图5.1表示了一个 struct 如何划分为对象和内存位置。

首先，整个 struct 是一个对象，它由几个子对象组成，各子对象对应每个数据成员。位域 bf1 和 bf2 共享一个内存位置，而 std::string 对象在内部由几个内存位置组成，但是其余的每个成员都有自己的内存位置。注意零长度的位域 bf3 是如何将 bf4 分割进它自己的内存位置的。

可以从中汲取四个要点。

- 每个变量都是一个对象，包括其他对象的成员。
- 每个对象占据至少一个内存位置。
- 如 int 或 char 这样的基本类型的变量恰好一个内存位置，不论其大小，即便它们相邻或是数组的一部分。
- 相邻的位域是相同内存位置的一部分。

我可以肯定你在怀疑这与并发有什么关系，那么让我们来看一看。

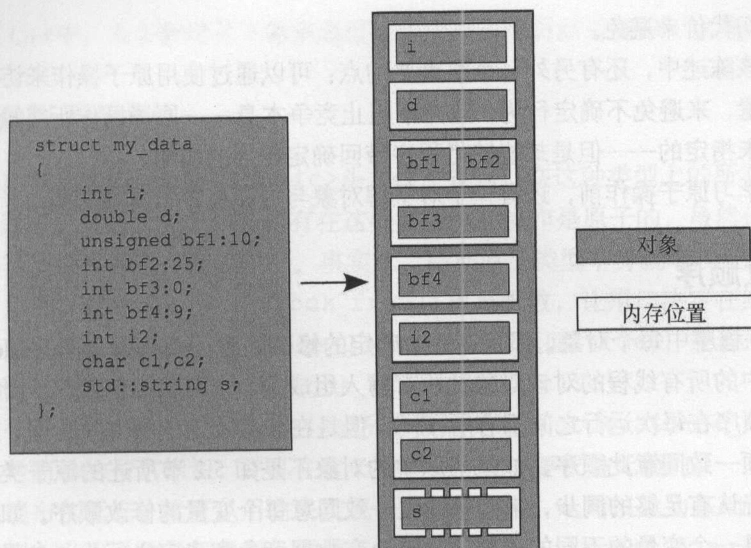



图 5.1 将 struct 划分为对象和内存位置

## 5.1.2 对象、内存位置以及并发

好了，这里是对于 C++中多线程应用程序至关重要的部分，所有东西都取决于这些内存位置。如果两个线程访问不同的内存位置，是没有问题的，一切工作正常。另一方面，如果两个线程访问相同的内存位置，那么你就得小心了。如果线程都没有在更新该内存位置，没问题，只读数据不需要进行保护或同步。如果任意一个线程正修改数据，就会有竞争条件的可能，如第 3 章所述。

为了避免竞争条件，在这两个线程的访问中间，就必须有一个强制的顺序。确保有一个确定顺序的方法之一，是使用如第 3 章中所述的互斥元。如果同一个互斥元在两个访问之前被锁定，则每次只有一个线程可以访问该内存位置，即有一个必然发生在另一个之前。另一种方法是在相同的或是其他的内存位置使用**原子**（**atomic**）操作的同步特性（参见 5.2 节对原子操作的定义），在两个线程的访问之间强加一个顺序。用原子操作来强制顺序描述于 5.3 节。如果多于两个线程访问同一个内存位置，则每一对访问都必须具有明确的顺序。

如果来自独立线程的两个对同一内存位置的访问没有强制顺序，其中一个或两个访问不是原子的，且一个或两个是写操作，那么这就是数据竞争并导致未定义行为。

这个陈述是至关重要的：未定义行为是 C++最令人不爽的状况之一。根据语言标准，一旦应用程序包含未定义行为，那么一切都很难说，整个应用程序的行为都是未定义的，它能干出任何事情来。我所知道的一个情况是，某个未定义行为的实例导致某人的监视器着火了。虽然这不大可能发生在你身上，但数据竞争绝对是一个严重的错误，且应该

不惜一切代价来避免。

在该陈述中，还有另外一个很重要的点：可以通过使用原子操作来访问具有竞争的内存位置，来避免不确定行为。这并不阻止竞争本身——原子操作所接触的内存位置首先仍是未指定的——但是却能够将程序带回确定行为的领域。

在学习原子操作前，还有一个对了解对象与内存位置很重要的概念：修改顺序。

### 5.1.3　修改顺序

C++程序中每个对象，都具有一个确定的**修改顺序**（**modification order**），它是由来自程序中的所有线程的对该对象的所有写入组成的，由对象的初始化开始。在多数情况下，该顺序在每次运行之间都有所不同，但是在任意给定的程序执行里，系统中的所有线程必须一致同意此顺序。如果问题中的对象不是如5.2节所述的原子类型之一，你就得负责确认有足够的同步，来确保线程一致同意每个变量的修改顺序。如果不同的线程看到的是一个变量的不同的顺序值，就会有数据竞争和未定义行为（参见5.1.2节）。如果使用原子操作，编译器将负责确保必要的同步已就位。

这一要求意味着某些投机性的执行是禁止的，因为一旦线程已在修改顺序中看到了某个特定项，则后续读取操作必须返回更晚的值，同时来自该线程对此对象后续的写入操作在修改顺序中也必须发生得更晚。同样，在同一线程中，在对象的写操作之后的读取，就必须返回要么写入的值，要么在该对象的修改顺序中更晚发生的另一个值。尽管所有的线程都必须一致同意程序中每一个独立对象的修改顺序，但却不必一致同意不同对象上操作的相对顺序。参见5.3.3节更多地了解线程间的操作顺序。

那么，是什么构成了原子操作，它又是如何用来强制顺序的？

## 5.2　C++中的原子操作及类型

**原子操作**（**atomic operation**）是一个不可分割的操作。从系统中的任何一个线程中，你都无法观察到一个完成到一半的这种操作，它要么做完了，要么就没做完。如果读取对象值的载入操作是**原子的**（**atomic**），并且所有对该对象的修改也都是**原子的**，那么这个载入操作所获取到的要么是对象的初始值，要么是被某个修改者存储后的值。

其对立面，是一个非原子操作可能被另一个线程视为半完成的。如果这是一次存储操作，那么被另一个线程观察到的值可能既不是储存前的值也不是已存储的值，而是其他的东西。如果执行非原子的载入操作，那么它可能获取对象的一部分，由另一个线程修改了值，然后再获取其余的对象，这样获取到的既不是第一个值也不是第二个值，而是两者的某种组合。这就是一个简单的有问题的竞争条件，正如第3章所述的那样，但是在这个级别，它可能构成一个**数据竞争**（参见5.1节）并因而导致未定义行为。

在 C++中，大多数情况下需要通过原子类型来得到原子操作，让我们来看一看。

### 5.2.1 标准原子类型

标准原子类型可以在<atomic>头文件中找到。在这种类型上的所有操作都是原子的，在语言定义的意义上说，只有在这些类型上的操作是原子的，虽然你可以利用互斥元使得其他操作看起来像原子的。事实上，标准原子类型本身就可以进行这样的模拟，它们（几乎）都具有一个 is_lock_free()成员函数，让用户决定在给定类型上的操作是否直接用原子指令完成（x.is_lock_free()返回 true），或者通过使用编译器和类库内部的锁来完成（x.is_lock_free()返回 false）。

唯一不提供 is_lock_free()成员函数的类型是 std::atomic_flag。该类型是一个非常简单的布尔标识，并且在这个类型上的操作要求是无锁的。一旦你有了一个简单的无锁布尔标志，就可以用它来实现一个简单的锁，从而以此为基础实现所有其他的原子类型。当我说非常简单时，指的是：类型为 std::atomic_flag 的对象被初始化为清除，接下来，它们可以被查询和设置（通过 test_and_set()成员函数）或者清除（通过 clear()成员函数）。就是这样：没有赋值，没有拷贝构造函数，没有测试和清除，也没有任何其他的操作。

其余的原子类型全都是通过 std::atomic<>类模板的特化来访问的，并且有一点更加的全功能，但可能不是无锁的（如前所述）。在大多数流行的平台上，我们认为内置类型的原子变种（例如 std::atomic<int>和 std::atomic<void*>）确实是无锁的，但却并非是要求的。你马上会看到，每个特化的接口反映了类型的属性。例如，像&=这样的位操作没有为普通指针作定义，因此它们也没有为原子指针作定义。

除了可以直接使用 std::atomic<>，你还可以使用表 5.1 所示的一组名称，来提及已经提供实现的原子类型。鉴于原子类型如何被添加到 C++标准的历史，这些替代类型名称可能指的是相应的 std::atomic<>特化，也可能是该特化的基类。在同一个程序中混用这些替代名称和 std::atomic<>特化的直接命名，可能会因此导致不可移植的代码。

表 5.1 标准原子类型的替代名称和它们所对应的 std::atomic<>特化

| 原子类型 | 对应的特化 |
|---|---|
| atomic_bool | std::atomic<bool> |
| atomic_char | std::atomic<char> |
| atomic_schar | std::atomic<signed char> |
| atomic_uchar | std::atomic<unsigned char> |
| atomic_int | std::atomic<int> |
| atomic_uint | std::atomic<unsigned> |
| atomic_short | std::atomic<short> |

续表

| 原子类型 | 对应的特化 |
|---|---|
| atomic_ushort | std::atomic<unsigned short> |
| atomic_long | std::atomic<long> |
| atomic_ulong | std::atomic<unsigned long> |
| atomic_llong | std::atomic<long long> |
| atomic_ullong | std::atomic<unsigned long long> |
| atomic_char16_t | std::atomic<char16_t> |
| atomic_char32_t | std::atomic<char32_t> |
| atomic_wchar_t | std::atomic<wchar_t> |

同基本原子类型一样，C++标准库也为原子类型提供了一组 typedef，对应于各种像 std::size_t 这样的非原子的标准库 typedef，如表 5.2 所示。

表 5.2　标准库原子类型定义以及其对应的内置类型定义

| 原子 typedef | 对应的标准库 typedef |
|---|---|
| atomic_int_least8_t | int_least8_t |
| atomic_uint_least8_t | uint_least8_t |
| atomic_int_least16_t | int_least16_t |
| atomic_uint_least16_t | uint_least16_t |
| atomic_int_least32_t | int_least32_t |
| atomic_uint_least32_t | uint_least32_t |
| atomic_int_least64_t | int_least64_t |
| atomic_uint_least64_t | uint_least64_t |
| atomic_int_fast8_t | int_fast8_t |
| atomic_uint_fast8_t | uint_fast8_t |
| atomic_int_fast16_t | int_fast16_t |
| atomic_uint_fast16_t | uint_fast16_t |
| atomic_int_fast32_t | int_fast32_t |
| atomic_uint_fast32_t | uint_fast32_t |
| atomic_int_fast64_t | int_fast64_t |
| atomic_uint_fast64_t | uint_fast64_t |
| atomic_intptr_t | intptr_t |
| atomic_uintptr_t | uintptr_t |
| atomic_size_t | size_t |
| atomic_ptrdiff_t | ptrdiff_t |
| atomic_intmax_t | intmax_t |
| atomic_uintmax_t | uintmax_t |

好多的类型啊！有一个很简单的模式，对于标准的 typedef T，其对应的原子类型与之同名并带有 atomic_前缀：atomic_T。这同样适用于内置类型，区别是 signed 缩写为 s，unsigned 缩写为 u，long long 缩写为 llong。一般来说，无论你要使用的是什么 T，都只是简单地说 std::atomic<T>，而不是使用替代名称。

传统意义上，标准原子类型是不可复制且不可赋值的，因为它们没有拷贝构造函数和拷贝赋值运算符。但是，它们确实支持从相应的内置类型的赋值进行隐式转换并赋值，与直接 load()和 store()成员函数、exchange()、compare_exchange_weak()以及 compare_exchange_strong()一样。它们在适当的地方还支持复合赋值运算符：+=、-=、*=、|=等，同时整型和针对指针的 std::atomic<>特化还支持++和--。这些运算符也拥有相应命名的具有相同功能的成员函数：fetch_add()、fetch_or()等。赋值运算和成员函数的返回值既可以是存储的值（在赋值运算符的情况下）或运算之前的值（在命名函数的情况下）。这避免了可能出现的问题，这些问题源于这种赋值运算符通常会返回一个将要赋值的对象的引用。为了从这种引用中获取存储的值，代码就得执行独立的读操作，这就允许另一个线程在赋值运算和读操作之间修改其值，并为竞争条件敞开大门。

然而，std::atomic<>类模板并不仅仅是一组特化。它具有一个主模板，可以用于创建一个用户定义类型的原子变种。由于它是一个泛型类模板，操作只限为 load()、store()（和与用户定义类型间的相互赋值）、exchange()、compare_exchange_weak()和 compare_exchange_strong()。

在原子类型上的每一个操作均具有一个可选的内存顺序参数，它可以用来指定所需的内存顺序语义。内存顺序选项的确切语义参见 5.3 节。就目前而言，了解运算分为三种类型就足够了。

- 存储（**store**）操作，可以包括 memory_order_relaxed、memory_order_release 或 memory_order_seq_cst 顺序
- 载入（**load**）操作，可以包括 memory_order_relaxed、memory_order_consume、memory_order_acquire 或 memory_order_seq_cst 顺序
- 读-修改-写（**read-modify-write**）操作，可以包括 memory_order_relaxed、memory_order_consume、memory_order_acquire、memory_order_release、memory_order_acq_rel 或 memory_order_seq_cst 顺序

所有操作的默认顺序为 memory_order_seq_cst。

现在让我们来看看能够在标准原子类型上进行的实际操作，从 std::atomic_flag 开始。

## 5.2.2 std::atomic_flag 上的操作

std::atomic_flag 是最简单的标准原子类型，它代表一个布尔标志。这一类型的对象可以是两种状态之一：设置或清除。这是故意为之的基础，仅仅是为了用作构造块。基于此，除非在极特殊情况下，否则我都不希望看到使用它。即便如此，它将作为讨论其他原子类型的起点，因为它展示了一些应用于原子类型的通用策略。

类型为 std::atomic_flag 的对象必须用 ATOMIC_FLAG_INIT 初始化。这会将该标志初始化为清除状态。在这里没有其他的选择，此标志总是以清除开始。

```
std::atomic_flag f=ATOMIC_FLAG_INIT;
```

这适用于所有对象被声明的地方，且无论其具有什么作用域。这是唯一需要针对初始化进行特殊处理的原子类型，但同时也是唯一保证无锁的类型。如果 std::atomic_flag 对象具有状态存储持续时间，那么就保证了静态初始化，这意味着不存在初始化顺序问题，它总是在该标识上的首次操作时进行初始化。

一旦标识对象初始化完成，你只能对它做三件事：销毁、清除或设置并查询其先前的值。这些分别对应于析构函数、clear()成员函数以及 test_and_set()成员函数。clear()和 test_and_set()成员函数都可以指定一个内存顺序。clear()是一个存储操作，因此不能有 memory_order_acquire 或 memory_order_acq_rel 语义，但是 test_and_set()是一个读-修改-写操作，并因此能够适用任意的内存顺序标签。至于每个原子操作，其默认值都是 memory_order_seq_cst。例如，

```
f.clear(std::memory_order_release);     <-❶
bool x=f.test_and_set();                <-❷
```

此处，对 clear()的调用❶明确要求使用释放语义清除该标志，而对 test_and_set()的调用❷使用默认内存顺序来设置标志和获取旧的值。

你不能从一个 std::atomic_flag 对象拷贝构造另一个对象，也不能将一个 std::atomic_flag 赋值给另外一个。这并不是 std::atomic_flag 特有的，而是所有原子类型共有的。在原子类型上的操作全都定义为原子的，以及包括两个对象的赋值和拷贝构造。在两个不同对象上的单一操作不可能是原子的。在拷贝构造或拷贝赋值的情况下，其值必须先从一个对象读取，再写入另外一个。这是在两个独立对象上的独立操作，其组合不可能是原子的。因此，这些操作是禁止的。

有限的特性集使得 std::atomic_flag 理想地适合于用作自旋锁互斥元。最开始，该标志置为清除，互斥元是解锁的。为了锁定互斥元，循环执行 test_and_set()直至旧值为 false，指示这个线程将值设为 true。解锁此互斥元就是简单地清除标志。这个实现展示在清单 5.1 中。

**清单 5.1 使用 std::atomic_flag 的自旋锁互斥实现**

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex():
        flag(ATOMIC_FLAG_INIT)
    {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }
```

```
    void unlock()
    {
        flag.clear(std::memory_order_release);
    }
};
```

这样一个互斥元是非常基本的，但它足以与 std::lock_guard<>一同使用（参见第 3 章）。就其本质而言，它在 lock()中执行了一个忙等待，因此如果你希望有任何程度的竞争，这就是个糟糕的选择，但它足以确保互斥。当我们观察内存顺序语义时，将看到这是如何保证与互斥元锁相关的必要的强制顺序。这个例子将在 5.3.6 节中阐述。

std::atomic_flag 由于限制性甚至不能用作一个通用的布尔标识，因为它不具有简单的无修改查询操作。为此，你最好还是使用 std::atomic<bool>，接下来我将介绍之。

### 5.2.3 基于 std::atomic<bool>的操作

最基本的原子整数类型是 std::atomic<bool>。如你所期望那样，这是一个比 std::atomic_flag 功能更全的布尔标志。虽然它仍然是不可拷贝构造和拷贝赋值的，但可以从一个非原子的 bool 来构造它，所以它可以被初始化为 true 或 false，同时也可以从一个非原子的 bool 值来对 std::atomic<bool>的实例赋值。

```
std::atomic<bool> b(true);
b=false;
```

从非原子的 bool 进行赋值操作还要注意的一件事是它与通常的惯例不同，并非向其赋值的对象返回一个引用，它返回的是具有所赋值的 bool。这对于原子类型是另一种常见的模式，它们所支持的赋值操作符返回值（属于相应的非原子类型）而不是引用。如果返回的是原子变量的引用，所有依赖于赋值结果的代码将显式地载入该值，可能会获取被另一线程修改的结果。通过以非原子值的形式返回赋值结果，可以避免这种额外的载入，你会知道所获取到的值是已存储的实际值。

与使用 std::atomic_flag 的受限的 clear()函数不同，写操作（无论是 true 还是 false）是通过调用 store()来完成的。类似地，test_and_set()被更通用的 exchange()成员函数所替代，它可以让你用所选择的新值来代替已存储的值，同时获取原值。std::atomic<bool>支持对值的普通无修改查询，通过隐式转化成普通的 bool，或显式调用 load()。正如你所期望的，store()是一个存储操作，而 load()是载入操作，exchange()是读-修改-写操作。

```
std::atomic<bool> b;
bool x=b.load(std::memory_order_acquire);
b.store(true);
x=b.exchange(false,std::memory_order_acq_rel);
```

exchange()并非 std::atomic<bool>支持的唯一的读-修改-写操作，它还引入

了一个操作，用于在当前值与期望值相等时，存储新的值。

## 根据当前值存储一个新值（或者否）

这个新的操作被称为比较/交换，它以 compare_exchange_weak()和 compare_exchange_strong()成员函数形式出现。比较/交换操作是使用原子类型编程的基石，它比较原子变量值和所提供的期望值，如果两者相等，则存储提供的期望值。如果两者不等，则期望值更新为原子变量的实际值。比较/交换函数的返回类型为 bool，如果执行了存储即为 true，反之则为 false。

对于 compare_exchange_weak()，即使原始值等于期望值也可能出现存储不成功，在这种情况下变量的值是不变的，compare_exchange_weak()的返回值为 false。这最有可能发生在缺少单个的比较并交换指令的机器上，此时处理器无法保证该操作被完成——可能因为执行操作的线程在必需的指令序列中间被切换出来，同时在线程多余处理器数量的操作系统中，它被另一个计划中的线程代替。这就是所谓的伪失败（**spurious failure**），因为失败的原因是时间的函数而不是变量的值。

由于 compare_exchange_weak()可能会伪失败，它通常必须用在循环中。

```
bool expected=false;
extern atomic<bool> b; // 在别处设置
while(!b.compare_exchange_weak(expected,true) && !expected);
```

在这种情况下，只要 expected 仍为 false，表明 compare_exchange_weak()调用伪失败，就保持循环。

另一方面，仅当实际值不等于 expected 值时 compare_exchange_strong()才保证返回 false。这样可以消除对循环的需求，正如你希望知道它所显示的那样，是否成功改变了一个变量，或者是否有另一个线程抢先抵达。

如果你想要改变变量，无论其初始值是多少（也许是用一个依赖于当前值的更新值），expected 的更新就变得很有用，每次经过循环时，expected 被重新载入，因此如果没有其他线程在此期间修改其值，那么 compare_exchange_weak()或 compare_exchange_strong()的调用在下一次循环中应该是成功的。如果计算待存储的值很简单，为了避免在 compare_exchange_weak()可能会伪失败的平台上的双重循环，（于是 compare_exchange_strong()包含一个循环），则使用 compare_exchange_weak()可能是有好处的。另一方面，如果计算待存储的值本身是耗时的，当 expected 值没有变化时，使用 compare_exchange_strong()来避免被迫重新计算待存储的值可能是有意义的。对于 std::atomic<bool>而言这并不重要——毕竟只有两个可能的值——但对于较大的原子类型这会有所不同。

比较/交换函数还有一点非同寻常，它们可以接受**两个**内存顺序参数。这就允许内存顺序的语义在成功和失败的情况下有所区别。对于一次成功调用具有

memory_order_acq_rel 语义而一次失败的调用有着 memory_order_relaxed 语义，这想必是极好的。一次失败的比较/交换并不进行存储，因此它无法具有 memory_order_release 或 memory_order_acq_rel 语义。因此在失败时，禁止提供这些值作为顺序。你也不应为失败提供比成功更严格的内存顺序。如果你希望 memory_order_acquire 或者 memory_order_seq_cst 作为失败的语义，你也必须为成功指定这些语义。

如果你没有为失败指定一个顺序，就会假定它与成功是相同的，除了顺序的 release 部分被除去：memory_order_release 变成 memory_order_relaxed，memory_order_acq_rel 变成 memory_order_acquire。如果你都没有指定，它们通常默认为 memory_order_seq_cst，这为成功和失败都提供了完整的序列顺序。以下对 compare_exchange_weak() 的两个调用是等价的。

```
std::atomic<bool> b;
bool expected;
b.compare_exchange_weak(expected,true,
    memory_order_acq_rel,memory_order_acquire);
b.compare_exchange_weak(expected,true,memory_order_acq_rel);
```

我将把选择内存顺序的后果留在 5.3 节。

std::atomic<bool>和 std::atomic_flag 之间的另外一个区别是 std::atomic<bool>可能不是无锁的。为了保证操作的原子性，实现可能需要内部地获得一个互斥锁。对于这种罕见的情况，当它重要时，你可以使用 is_lock_free() 成员函数检查是否对 std::atomic<bool>的操作是无锁的。除了 std::atomic_flag，对于所有原子类型，这是另一个特征共同。

原子类型中其次简单的是原子指针特化 std::atomic<T*>，那么接下来我们看看这些。

### 5.2.4 std::atomic<T*>上的操作：指针算术运算

对于某种类型 T 的指针的原子形式是 std::atomic<T*>，正如 bool 的原子形式是 std::atomic<bool>一样。接口基本上是相同的，只不过它对相应的指针类型的值进行操作而非 bool 值。与 std::atomic<bool>一样，它既不能拷贝构造，也不能拷贝赋值，虽然它可以从合适的指针值中构造和赋值。和必须的 is_lock_free() 成员函数一样，std::atomic<T*>也有 load()、store()、exchange()、compare_exchange_weak() 和 compare_exchange_strong() 成员函数，具有和 std::atomic<bool>相似的语义，也是接受和返回 T* 而不是 bool。

std::atomic<T*>提供的新操作是指针算术运算。这种基本的操作是由 fetch_add() 和 fetch_sub() 成员函数提供的，可以在所存储的地址上进行原子加法

和减法，+=、-=、带++和--的前缀与后缀的自增自减等运算符，都提供了方便的封装。运算符的工作方式，与你从内置类型中所期待的一样。如果 x 是 std::atomic<Foo*>指向 Foo 对象数组的第一项，那么 x+=3 将它改为指向第四项，并返回一个普通的指向第四项的 Foo*。fetch_add()和 fetch_sub()有细微的区别，它们返回的是原值（所以 x.fetch_add(3)将 x 更新为指向第四个值，但是返回一个指向数组中的第一个值的指针）。该操作也称为**交换与添加**，它是一个原子的读-修改-写操作，就像 exchange()和 compare_exchange_weak()/compare_exchange_strong()。与其他的操作一样，返回值是一个普通的 T*值而不是对 std::atomic<T*>对象的引用，因此，调用代码可以基于之前的值执行操作。

```
class Foo{};
Foo some_array[5];
std::atomic<Foo*> p(some_array);
Foo* x=p.fetch_add(2);              // 将 p 加 2 并返回原来的值
assert(x==some_array);
assert(p.load()==&some_array[2]);
x=(p-=1);                           // 将 p 减 1 并返回新的值
assert(x==&some_array[1]);
assert(p.load()==&some_array[1]);
```

该函数形式也允许内存顺序语义作为一个额外的函数调用参数而被指定。

```
p.fetch_add(3,std::memory_order_release);
```

因为 fetch_add()和 fetch_sub()都是读-修改-写操作，所以它们可以具有任意的内存顺序标签，也可以参与到**释放序列**中。对于运算形式，指定顺序语义是不可能的，因为没有办法提供信息，因此这些形式总是具有 memory_order_seq_cst 语义。

其余的基本原子类型本质上都是一样的，它们都是原子整型，并且彼此都具有相同的接口，区别是相关联的内置类型是不同的。我们将它们视为一个群组。

### 5.2.5 标准原子整型的操作

除了一组通常的操作（load()、store()、exchange()、compare_exchange_weak()和 compare_exchange_strong()）之外，像 std::atomic<int>或者 std::atomic<unsigned long long>这样的原子整型还有相当广泛的一组操作可用：fetch_add()、fetch_sub()、fetch_and()、fetch_or()、fetch_xor()，这些运算的复合赋值形式（+=、-=、&=、|=和^=），前缀/后缀自增和前缀/后缀自减（++x、x++、--x 和 x--）。这并不是很完整的一组可以在普通的整型上进行的复合赋值运算，但是已足够接近了，只有除法、乘法和位移运算符是缺失的。因为原子整型值通常作为计数器或者位掩码来使用，这不是一个特别明显的损失。如果需要的话，额外的操作可以通过在一个循环中使用 compare_exchange_weak()来实现。

这些语义和std::atomic<T*>的fetch_add()和fetch_sub()的语义密切地匹配，命名函数原子级执行它们的操作，并返回旧值，而复合赋值运算符返回新值。前缀与后缀自增自减也跟平常一样工作，++x 增加变量值并返回新值，而 x++ 增加变量并返回旧值。正如你到目前所期待的那样，在这两种情况下，结果都是一个相关联的整型值。

现在，我们已经了解了所有的基本原子类型，剩下的就是泛型 std::atomic<>初级类模板而不是这些特化，那么让我们接着看下面的内容。

### 5.2.6 std::atomic<>初级类模板

除了标准的原子类型，初级模板的存在允许用户创建一个用户定义的类型的原子变种。然而，你不能只是将用户定义类型用于 std::atomic<>，该类型必须满足一定的准则。为了对用户定义类型 UDT 使用 std::atomic<UDT>，这种类型必须有一个平凡的（**trivial**）拷贝赋值运算符。这意味着该类型不得拥有任何虚函数或虚基类，并且必须使用编译器生成的拷贝赋值运算符。不仅如此，一个用户定义类型的每个基类和非静态数据成员也都必须有一个平凡的拷贝赋值运算符。这实质上允许编译器将 memcpy()或一个等价的操作用于赋值操作，因为没有用户编写的代码要运行。

最后，该类型必须是**按位相等可比较**的。这伴随着赋值的要求，你不仅要能够使用 memcpy()复制 UDT 类型的对象，而且还必须能够使用 memcmp()比较实例是否相等。为了使比较/交换操作能够工作，这个保证是必需的。

这些限制的原因可以追溯到第 3 章中的一个准则，不要在锁定范围之外，向受保护的数据通过将其作为参数传递给用户所提供的函数的方式，传递指针和引用。一般情况下，编译器无法为 std::atomic<UDT>生成无锁代码，所以它必须对所有的操作使用一个内部锁。如果用户提供的拷贝赋值或比较运算符是被允许的，这将需要传递一个受保护数据的引用作为一个用户提供的函数的参数，因而违背准则。同样地，类库完全自由地为所有需要单个锁定的原子操作使用单个锁定，并允许用户提供的函数被调用，虽然认为因为一个比较操作花了很长时间，该锁可能会导致死锁或造成其他线程阻塞。最后，这些限制增加了编译器能够直接为 std::atomic<UDT>利用原子指令的机会（并因此使一个特定的实例无锁），因为它可以把用户定义的类型作为一组原始字节来处理。

需要注意的是，虽然你可以使用 std::atomic<float>或 std::atomic<double>，因为内置的浮点类型确实满足与 memcpy 和 memcmp 一同使用的准则，在 compare_exchange_strong 情况下这种行为可能会令人惊讶。如果存储的值具有不同的表示，即使旧的存储值与被比较的值的数值相等，该操作可能会失败。请注意，浮

点值没有原子的算术操作。如果你使用一个具有定义了相等比较运算符的用户定义类型的 std::atomic<>，你会得到和 compare_exchange_strong 类似的情况，而且该操作符会与使用 memcmp 进行比较不同——该操作可能因另一相等值具有不同的表示而失败。

如果你的 UDT 和一个 int 或一个 void*大小相同（或更小），大部分常见的平台能够为 std::atomic<UDT>使用原子指令。一些平台也能够使用大小是 int 或 void*两倍的用户定义类型的原子指令。这些平台通常支持一个与 compare_exchange_xxx 函数相对应的所谓的**双字比较和交换**（**double-word-compare-and-swap，DWCAS**）指令。正如你将在第 7 章中看到的，这种支持可以帮助写无锁代码。

这些限制意味着，例如，你不能创建一个 std::atomic<std::vector<int>>，但你可以把它与包含计数器、标识符、指针、甚至简单的数据元素的数组的类一起使用。这不是特别的问题。越是复杂的数据结构，你就越有可能想在它之上进行简单的赋值和比较之外的操作。如果情况是这样，你最好还是使用 std::mutex，来确保所需的操作得到适当的保护，正如在第 3 章中所描述的。

当使用一个用户定义的类型 T 进行实例化时，std::atomic<T>的接口被限制为一组操作，对于 std::atomic<bool>:load()、store()、exchange()、compare_exchange_weak()、compare_exchange_strong()，以及赋值自和转换到类型 T 的实例，是可用的。

表 5.3 展示了在每个原子类型上的可用操作。

表 5.3　原子类型的可用操作

| 操作 | atomic_flag | atomic<bool> | atomic<T*> | atomic <integral-type> | atomic <other-type> |
|---|---|---|---|---|---|
| test_and_set | ✓ | | | | |
| clear | ✓ | | | | |
| is_lock_free | | ✓ | ✓ | ✓ | ✓ |
| load | | ✓ | ✓ | ✓ | ✓ |
| store | | ✓ | ✓ | ✓ | ✓ |
| exchange | | ✓ | ✓ | ✓ | ✓ |
| compare_exchang e_weak, compare_exchange_strong | | ✓ | ✓ | ✓ | ✓ |
| fetch_add, += | | | ✓ | ✓ | |
| fetch_sub, -= | | | ✓ | ✓ | |
| fetch_or, \|= | | | | ✓ | |
| fetch_and, &= | | | | ✓ | |
| fetch_xor, ^= | | | | ✓ | |
| ++, -- | | | ✓ | ✓ | |

## 5.2.7 原子操作的自由函数

到现在为止，我一直限制自己去描述原子类型的操作的成员函数形式。然而，各种原子类型上的所有操作也都有等效的非成员函数。对于大部分非成员函数来说，都是以相应的成员函数来命名的，只是带有一个 atomic_前缀（例如 std::atomic_load()）。这些函数也为每个原子类型进行了重载。在有机会指定内存顺序标签的地方，它们有两个变种：一个没有标签，一个带有_explicit 后缀和额外的参数作为内存顺序的标签（例如，std::atomic_store(&atomic_var,new_value)与 std::atomic_store_explicit(&atomic_var,new_value,std::memory_order_release)相对）。被成员函数引用的原子对象是隐式的，然而所有的自由函数接受一个指向原子对象的指针作为第一个参数。

例如，std::atomic_is_lock_free()只有一个变种（尽管为每个类型重载），而 std::atomic_is_lock_free(&a)对于原子类型 a 的对象，与 a.is_lock_free()返回相同的值。同样地，std::atomic_load(&a)和 a.load()是一样的，但是 a.load(std::memory_order_acquire) 等同于 std::atomic_load_explicit(&a, std::memory_order_acquire)。

自由函数被设计为可兼容 C 语言，所以它们在所有情况下使用指针而不是引用。例如，compare_exchange_weak()和 compare_exchange_strong()成员函数（期望值）的第一参数是引用，而 std::atomic_compare_exchange_weak()（第一个参数是对象指针）的第二个参数是指针。std::atomic_compare_exchange_weak_explicit()同时也需要指定成功与失败的内存顺序。而比较/交换成员函数具有一个单内存顺序形式（默认为 std::memory_order_seq_cst）和一个分别接受成功与失败内存顺序的重载。

std::atomic_flag 上的操作违反这一潮流，原因在于它们在名字中拼出“flag”的部分：std::atomic_flag_test_and_set()、std::atomic_flag_clear()，尽管指定内存顺序的其他变种具有_explicit 后缀：std::atomic_flag_test_and_set_explicit()和 std::atomic_flag_clear_explicit()。

C++标准库还提供了为了以原子行为访问 std::shared_ptr<>实例的自由函数。这打破了只有原子类型支持原子操作的原则，因为 std::shared_ptr<>很肯定地不是原子类型。然而，C++标准委员会认为提供这些额外的函数是足够重要的。可用的原子操作有：**载入（load）**、**存储（store）**、**交换（exchange）**和**比较/交换（compare/exchange）**，这些操作以标准原子类型上的相同操作的重载的形式被提供，接受 std::shared_ptr<>*作为第一个参数。

```
std::shared_ptr<my_data> p;
void process_global_data()
{
    std::shared_ptr<my_data> local=std::atomic_load(&p);
    process_data(local);
}
void update_global_data()
{
    std::shared_ptr<my_data> local(new my_data);
    std::atomic_store(&p,local);
}
```

至于其他类型上的原子操作，_explicit 变量也被提供并允许你指定所需的内存顺序，std::atomic_is_lock_free()函数可以用于检查是否实现使用了锁以确保原子性。

正如在引言中所提到的，标准的原子类型能做的不仅仅是避免与数据竞争相关的未定义行为，它们允许用户在线程间强制操作顺序。这个强制的顺序是为保护数据和诸如 std::mutex 和 std::future<>这样同步操作的工具的基础。考虑到这一点，让我们进入到本章真正的重点，内存模型的并发方面的细节，以及原子操作如何用来同步数据和强制顺序。

## 5.3 同步操作和强制顺序

假设有两个线程，其中一个是要填充由第二个线程所读取的数据结构。为了避免有问题的竞争条件，第一个线程设置一个标志来指示数据已经就绪，在标志设置之前第二个线程不读取数据。清单 5.2 展示了这样的场景。

**清单 5.2 从不同的线程中读取和写入变量**

```
#include <vector>
#include <atomic>
#include <iostream>

std::vector<int> data;
std::atomic<bool> data_ready(false);
void reader_thread()
{
    while(!data_ready.load())    <-❶
    {
        std::this_thread::sleep(std::milliseconds(1));
    }
    std::cout<<"The answer="<<data[0]<<"\n";    <-❷
}
void writer_thread()
{
    data.push_back(42);    <-❸
    data_ready=true;    <-❹
}
```

撇开等待数据准备的循环的低效率❶，你确实需要这样工作，因为不这样的话在线程间共享数据就变得不可行，每一项数据都强制成为原子的。你已经知道在没有强制顺序的情况下，非原子的读❷和写❸同一个数据是未定义的行为，所以为了使其工作，某个地方必须有一个强制的顺序。

所要求的强制顺序来自于 `std::atomic<bool>`变量 `data_ready` 上的操作，它们通过 **happens-before**（发生于之前）和 **synchronizes-with**（与之同步）内存模型关系的优点来提供必要的顺序。数据写入❸发生于写入 `data_ready` 标志❹之前，标志的读取❶发生于数据的读取❷之前。当从 `data_ready`❶读取的值为 `true` 时，写与读同步，创建一个 happens-before 的关系。因为 happens-before 是可传递的，数据的写入❸发生于标志的写入❹之前，发生于从标志读取 `true` 值❶之前，又发生于数据的读取❷之前，你就有了一个强制顺序：数据的写入发生于数据的读取之前，一切都好了。图 5.2 表明了两个线程间 happens-before 关系的重要性。我从读线程添加了 `while` 循环的几次迭代。

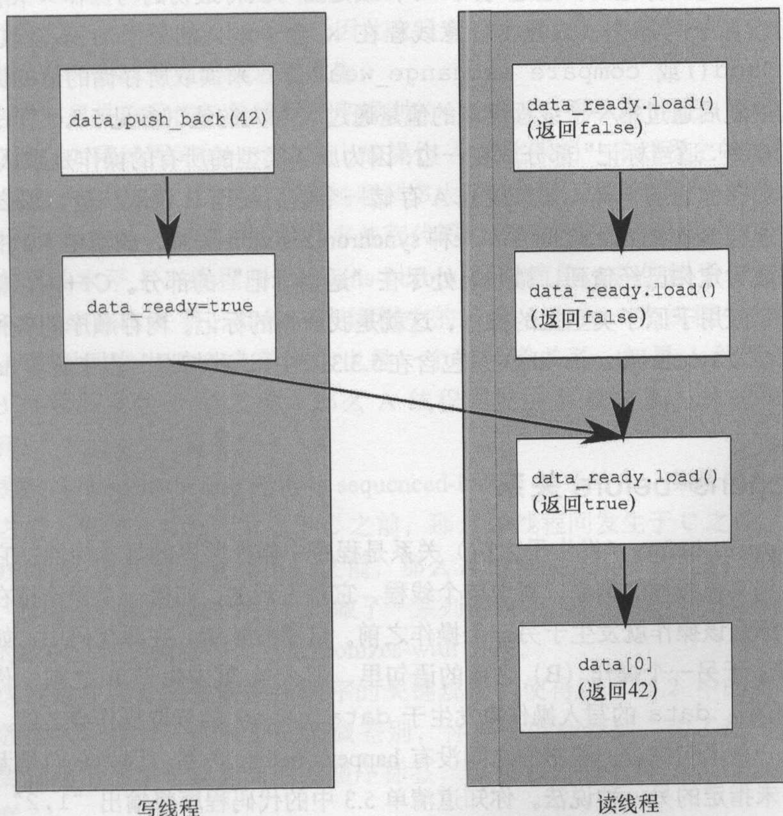



图 5.2 用原子操作在非原子操作之间强制顺序

所有这一切都似乎相当直观，写入一个值当然发生在读取这个值之前！使用默认的原子操作，这的确是真的（这就是为什么它是默认的），但它需要阐明：原子操作对于顺序要求也有其他的选项，这一点我马上会提到。

现在，你已经在实战中看到了 happens-before 和 synchronizes-with，现在是时候来看看它们到底是什么意思了。我将从 synchronizes-with 开始。

## 5.3.1 synchronizes-with 关系

synchronizes-with 关系是你只能在原子类型上的操作之间得到的东西。如果一个数据结构包含原子类型，并且在该数据结构上的操作会在内部执行适当的原子操作，该数据结构上的操作（如锁定互斥元）可能会提供这种关系，但是从根本上说 synchronizes-with 关系只出自原子类型上的操作。

基本的思想是这样的：在一个变量 `x` 上的一个被适当标记的原子写操作 `W`，与在 `x` 上的一个被适当标记的，通过写入(`W`)，或是由与执行最初的写操作 `W` 相同的线程在 `x` 上的后续原子写操作，或是由任意线程在 `x` 上一系列的原子的读-修改-写操作（如 `fetch_add()`或 `compare_exchange_weak()`）来读取所存储的值的原子读操作同步，其中随后通过第一个线程读取的值是通过 `W` 写入的值（参见 5.3.4 节）。

现在将“适当标记”部分放在一边，因为原子类型的所有的操作是默认适当标记的。这基本上和你想的一样：如果线程 A 存储一个值而线程 B 读取该值，那么线程 A 中的存储和线程 B 中的载入之间存在一种 synchronizes-with 关系，如清单 5.2 中一样。

我敢肯定你已经猜到，微妙之处尽在“适当标记”的部分。C++内存模型允许各种顺序约束应用于原子类型上的操作，这就是我所指的标记。内存顺序的各种选项以及它们如何涉及 synchronizes-with 关系包含在 5.3.3 节中。让我们退一步来看看 happens-before 关系。

## 5.3.2 happens-before 关系

**happens-before**（发生于之前）关系是程序中操作顺序的基本构件，它指定了哪些操作看到其他操作的结果。对于单个线程，它是直观的，如果一个操作排在另一个操作之前，那么该操作就发生于另一个操作之前。这就意味着，在源代码中，如果一个操作（A）发生于另一个操作（B）之前的语句里，那么 A 就发生于 B 之前。你可以在清单 5.2 中看到：`data` 的写入操作❸发生于 `data_ready` 的读取操作❹之前。如果操作发生在同一条语句中，一般它们之间没有 happens-before 关系，因为它们是无序的。这只是顺序未指定的另一种说法。你知道清单 5.3 中的代码程序将输出“`1,2`”或“`2,1`”，但是并未指定究竟是哪个，因为对 `get_num()`的两次调用的顺序未指定。

**清单 5.3 一个函数调用的参数的估计顺序是未指定的**

```
#include <iostream>

void foo(int a,int b)
{
    std::cout<<a<<","<<b<<std::endl;
}

int get_num()
{
    static int i=0;
    return ++i;
}

int main()
{
    foo(get_num(),get_num());    <-- 对 get_num()的调用是无序的
}
```

有时候，单条语句中的操作是有顺序的，例如使用内置的逗号操作符或者使用一个表达式的结果作为另一个表达式的参数。但是，一般来说，单条语句中的操作是非顺序的，而且也没有 sequenced-before（因此也没有 happens-before）。当然，一条语句中的所有操作在下一句的所有操作之前发生。

这确实只是你习惯的单线程顺序规则的一个重述，那么新的呢？新的部分是线程之间的交互，如果线程间的一个线程上的操作 A 发生于另一个线程上的操作 B 之前，那么 A 发生于 B 之前。这并没有真的起到多大帮助，你只是增加了一种新的关系（线程间 happens-before），但这在你编写多线程代码时是一个重要的关系。

在基础水平上，线程间 happens-before 相对简单，并依赖于 5.3.1 节中介绍的 synchronizes-with 关系，如果一个线程中的操作 A 与另一个线程中的操作 B 同步，那么 A 线程间发生于 B 之前。这也是一个可传递关系，如果 A 线程间发生于 B 之前，B 线程间发生于 C 之前，那么 A 线程间发生于 C 之前。你也可以在清单 5.2 中看到。

线程间 happens-before 还可与 sequenced-before 关系结合，如果操作 A 的顺序在操作 B 之前，操作 B 线程间发生于 C 之前，那么 A 线程间发生于 C 之前。类似地，如果 A 与 B 同步，B 的顺序在操作 C 之前，那么 A 线程间发生于 C 之前。这两个在一起意味着，如果你对单个线程中的数据做了一系列的改变，对于执行 C 的线程上的后续线程可见的数据，你只需要一个 synchronizes-with 关系。

这些是在线程间强制操作顺序的关键规则，使得清单 5.2 中的所有东西得以运行。数据依赖性有一些额外的细微差别，你很快就会看到。为了让你理解这一点，我需要介绍用于原子操作的内存顺序标签，以及它们如何与 synchronizes-with 关系相关联。

### 5.3.3 原子操作的内存顺序

有六种内存顺序选项可以应用到原子类型上的操作：memory_order_relaxed、memory_order_consume、memory_order_acquire、memory_order_release、emory_order_acq_rel 和 memory_order_seq_cst。除非你为某个特定的操作作出指定，原子类型上的所有操作的内存顺序选项都是 memory_order_seq_cst，这是最严格的可用选项。尽管有六种顺序选项，他们其实代表了三种模型:**顺序一致**（**sequentially consistent**）顺序（memory_order_seq_cst）、**获得-释放**（**acquire-release**）顺序(memory_order_consume、memory_order_acquire、memory_order_release 和 memory_order_acq_rel)，以及**松散**（**relaxed**）顺序(memory_order_relaxed)。

这些不同的内存顺序模型在不同的 CPU 架构上可能有着不同的成本。例如，在基于具有通过处理器而非做更改者对操作的可见性进行良好控制架构上的系统中，顺序一致的顺序相对于获得-释放顺序或松散顺序，以及获得-释放顺序相对于松散顺序，可能会要求额外的同步指令。如果这些系统拥有很多处理器，这些额外的同步指令可能占据显著的时间量，从而降低该系统的整体性能。另一方面，为了确保原子性，对于超出需要的获得-释放排序，使用 x86 或 x86-64 架构（如在台式 PC 中常见的 Intel 和 AMD 处理器）的 CPU 不会要求额外的指令，甚至对于载入操作，顺序一致顺序不需要任何特殊的处理，尽管在存储时会有一点额外的成本。

不同的内存顺序模型的可用性，允许高手们利用更细粒度的顺序关系来提升性能，在不太关键的情况下，当允许使用默认的顺序一致顺序时，他们是有优势的。

为了选择使用哪个顺序模型，或是为了理解使用不同模型的代码中的顺序关系，了解该选择会如何影响程序行为是很重要的。因此让我们来看一看每种选择对于操作顺序和 synchronizes-with 的后果。

#### 1. 顺序一致顺序

默认的顺序被命名为**顺序一致**（**sequentially consistent**），因为这意味着程序的行为与一个简单的顺序的世界观是一致的。如果所有原子类型实例上的操作是顺序一致的，多线程程序的行为，就好像是所有这些操作由单个线程以某种特定的顺序进行执行一样。这是迄今为止最容易理解的内存顺序，这也是它作为默认值的原因。所有线程都必须看到操作的相同顺序。这使得推断用原子变量编写的代码的行为变得容易。你可以写下不同线程的所有可能的操作顺序，消除那些不一致的，并验证你的代码在其他程序里是否和预期的行为一样。这也意味着，操作不能被重排。如果你的代码在一个线程中有一个操作在另一个之前，其顺序必须对所有其他的线程可见。

从同步的观点来看，顺序一致的存储与读取该存储值的同一个变量的顺序一致载入

是同步的。这提供了一种两个（或多个）线程操作的顺序约束，但顺序一致比它更加强大。在使用顺序一致原子操作的系统中，所有在载入后完成的顺序一致原子操作，也必须出现在其他线程的存储之后。清单 5.4 中的示例实际演示了这个顺序约束。该约束并不会推进使用具有松散内存顺序的原子操作，它们仍然可以看到操作处于不同的顺序，所以你必须在所有的线程上使用顺序一致的操作以获利。

然而，易于理解就产生了代价。在一个带有许多处理器的弱顺序机器上，它可能导致显著的性能惩罚，因为操作的整体顺序必须与处理器之间保持一致，可能需要处理器之间进行密集（且昂贵）的同步操作。这就是说，有些处理器架构（如常见的 x86 和 x86-64 架构）提供相对低廉的顺序一致性，因此如果你担心使用顺序一致顺序对性能的影响，检查你的目标处理器架构的文档。

清单 5.4 实际展示了顺序一致性。x 和 y 的载入和存储是用 memory_order_seq_cst 显式标记的，尽管此标记在这种情况下可以省略，因为它是默认的。

**清单 5.4 顺序一致隐含着总体顺序**

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
    x.store(true,std::memory_order_seq_cst);          <-❶
}

void write_y()
{
    y.store(true,std::memory_order_seq_cst);          <-❷
}

void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst))             <-❸
        ++z;
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst))             <-❹
        ++z;
}

int main()
{
    x=false;
```

```
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);     ◁—❺
}
```

assert❺可能永远不触发，因为对 x 的存储❶或者对 y 的存储❷必须先发生，尽管没有特别指定。如果 read_x_then_y 中载入 y❸返回 false，x 的存储必须发生于 y 的存储之前，这时 read_y_then_x 中载入 x❹必须返回 true，因为 while 循环在这一点上确保 y 是 true。由于 memory_order_seq_cst 的语义需要在所有标记 memory_order_seq_cst 的操作上有着单个总体顺序，返回 false 的载入 y❸和对 x 的存储❶之间有一个隐含的顺序关系。为了有单一的总体顺序，如果一个线程看到 x==true，随后看到 y==false，这意味着在这个总体顺序上，x 的存储发生在 y 的存储之前。

当然，因为一切是对称的，它也可能反方向发生，x 的载入❹返回 false，迫使 y 的载入❸返回 true。在这两种情况下，z 都等于 1。两个载入都可能返回 true，导致 z 等于 2，但是无论如何 z 都不可能等于 0。

对于 read_x_then_y 看到 x 为 true 且 y 为 false 的情况，其操作和 happens-before 关系如图 5.3 所示。从 read_x_then_y 中 y 的载入到 write_y 中 y 的存储的虚线，展示了所需要的隐含的顺序关系以保持顺序一致。在 memory_order_seq_cst 操作的全局顺序中，载入必须发生在存储之前，以达到这里所给的结果。

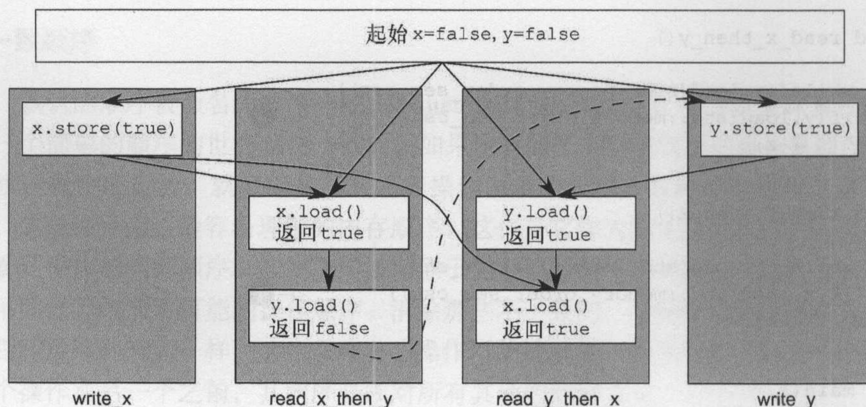



图 5.3　顺序一致和 happens-before

顺序一致是最直观和直觉的排序，但也是最昂贵的内存顺序，因为它要求所有线程之间的全局同步。在多处理器系统中，这可能需要处理器之间相当密集和耗时的通信。

为了避免这种同步成本，你需要跨出顺序一致的世界，并考虑使用其他的内存顺序。

### 2. 非顺序一致的内存顺序

一旦你走出美好的顺序一致的世界，事情开始变得复杂起来。可能要面对的一个最大问题是**事件不再有单一的全局顺序**的事实。这意味着，不同的线程可能看到相同的操作的不同方面，以及你所拥有的不同线程操作一前一后整齐交错的所有心理模型都必须扔到一边。你不仅得考虑事情真正的并行发生，而且**线程不必和事件的顺序一致**。为了编写（或者甚至只是理解）任何使用非默认的 memory_order_seq_cst 内存顺序的代码，让你的大脑思考这个问题绝对是至关重要的。这不仅仅意味着编译器能够重新排列指令。即使线程正在运行完全相同的代码，由于其他线程中的操作没有明确的顺序约束，它们可能与事件的顺序不一致，因为不同的 CPU 缓存和内部缓冲区可能为相同的内存保存了不同的值。它是如此重要以至于我要再说一遍：**线程不必和事件的顺序一致**。

你不仅要将基于交错操作的心理模型扔到一边，还得将基于编译器或处理器重排指令的思想的心理模型也扔掉。**在没有其他的顺序约束时，唯一的要求是所有的线程对每个独立变量的修改顺序达成一致**。不同变量上的操作可以以不同的顺序出现在不同的线程中，前提是所能看到的值与所有附加的顺序约束是一致的。

通过完全跳出顺序一致的世界，并未所有操作使用 memory_order_relaxed，就是最好的展示。一旦你掌握了，就可以回过头来看获得-释放顺序，它让你选择性地在操作之间引入顺序关系，夺回一些理性。

### 3. 松散顺序

以松散顺序执行的原子类型上的操作不参与 synchronizes-with 关系。单线程中的同一个变量的操作仍然服从 happens-before 关系，但相对于其他线程的顺序几乎没有任何要求。唯一的要求是，从同一个线程对单个原子变量的访问不能被重排，一旦给定的线程已经看到了原子变量的特定值，该线程之后的读取就不能获取该变量更早的值。在没有任何额外的同步的情况下，每个变量的修改顺序是使用 memory_order_relaxed 的线程之间唯一共享的东西。

为了展示松散顺序操作到底能多松散，你只需要两个线程，如清单 5.5 所示。

**清单 5.5 放松操作有极少数的排序要求**

```
#include <atomic>
#include <thread>
#include <assert.h>
```

```
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);     ❶

    y.store(true,std::memory_order_relaxed);     ❷
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));   ❸
    if(x.load(std::memory_order_relaxed))        ❹
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);     ❺
}
```

这次 assert❺ 可以触发，因为 x 的载入 ❹ 能够读到 false，即便是 y 的载入 ❸ 读到了 true 并且 x 的存储 ❶ 发生于 y 存储 ❷ 之前。x 和 y 是不同的变量，所以关于每个操作所产生的值的可见性没有顺序保证。

不同变量的松散操作可以被自由地重排，前提是它们服从所有约束下的 happens-before 关系（例如，在同一个线程中）。它们并不引入 synchronizes-with 关系。清单 5.5 中的 happens-before 关系如图 5.4 所示，伴随一个可能的结果。即便在存储操作之间和载入操作之间存在 happens-before 关系，但任一存储和任一载入之间却不存在，所以载入可以在顺序之外看到存储。

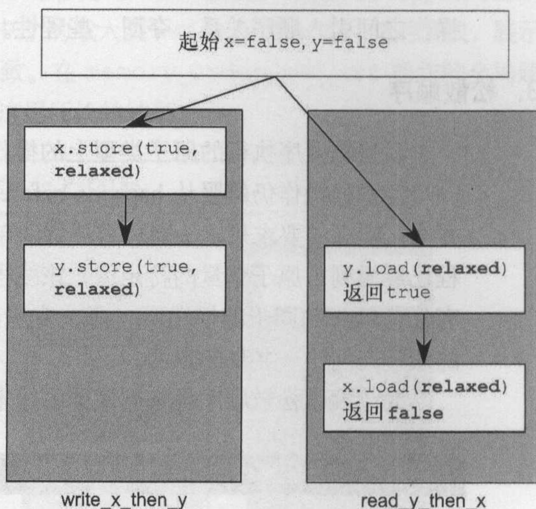



图 5.4　松散的原子和 happens-before

让我们在清单 5.6 中看一看带有三个变量和五个线程的稍微复杂的例子。

**清单 5.6 多线程的松散操作**

```
#include <thread>
#include <atomic>
#include <iostream>

std::atomic<int> x(0),y(0),z(0);    ←❶
std::atomic<bool> go(false);    ←❷

unsigned const loop_count=10;

struct read_values
{
    int x,y,z;
};

read_values values1[loop_count];
read_values values2[loop_count];
read_values values3[loop_count];
read_values values4[loop_count];
read_values values5[loop_count];

void increment(std::atomic<int>* var_to_inc,read_values* values)
{
    while(!go)
        std::this_thread::yield();    ←❸ 旋转，等待信号
    for(unsigned i=0;i<loop_count;++i)
    {
        values[i].x=x.load(std::memory_order_relaxed);
        values[i].y=y.load(std::memory_order_relaxed);
        values[i].z=z.load(std::memory_order_relaxed);
        var_to_inc->store(i+1,std::memory_order_relaxed);    ←❹
        std::this_thread::yield();
    }
}

void read_vals(read_values* values)
{
    while(!go)    ←❺ 旋转，等待信号
        std::this_thread::yield();
    for(unsigned i=0;i<loop_count;++i)
    {
        values[i].x=x.load(std::memory_order_relaxed);
        values[i].y=y.load(std::memory_order_relaxed);
        values[i].z=z.load(std::memory_order_relaxed);
        std::this_thread::yield();
    }
}

void print(read_values* v)
{
    for(unsigned i=0;i<loop_count;++i)
    {
        if(i)
```

```
            std::cout<<",";
        std::cout<<"("<<v[i].x<<","<<v[i].y<<","<<v[i].z<<")";
    }
    std::cout<<std::endl;
}

int main()
{
    std::thread t1(increment,&x,values1);
    std::thread t2(increment,&y,values2);
    std::thread t3(increment,&z,values3);
    std::thread t4(read_vals,values4);
    std::thread t5(read_vals,values5);

    go=true;              ⑥ 开始执行主循环的信号

    t5.join();
    t4.join();
    t3.join();
    t2.join();
    t1.join();
                          ⑦ 打印最终的值
    print(values1);
    print(values2);
    print(values3);
    print(values4);
    print(values5);
}
```

本质上这是一个非常简单的程序。你有三个共享的全局原子变量❶和五个线程。每个线程循环 10 次，用 memory_order_relaxed 读取三个原子变量的值，并将其存储在数组中。这三个线程中的每个线程每次通过循环❹更新其中一个原子变量，而其他两个线程只是读取。一旦所有的线程都被连接，你就打印由每个线程存储在数组中的值❼。

原子变量 go❷用来确保所有的线程尽可能靠近相同的时间开始循环。启动一个线程是一个昂贵的操作，若没有明确的延迟，第一个线程可能会在最后一个线程开始前就结束了。每个线程在进入主循环之前等待 go 变成 true❸、❺，仅当所有的线程都已经开始后❻，go 才被设置成 true。

这个程序的一个可能的输出如下所示。

```
(0,0,0),(1,0,0),(2,0,0),(3,0,0),(4,0,0),(5,7,0),(6,7,8),(7,9,8),(8,9,8),
    (9,9,10)
(0,0,0),(0,1,0),(0,2,0),(1,3,5),(8,4,5),(8,5,5),(8,6,6),(8,7,9),(10,8,9),
    (10,9,10)
(0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4),(0,0,5),(0,0,6),(0,0,7),(0,0,8),
    (0,0,9)
(1,3,0),(2,3,0),(2,4,1),(3,6,4),(3,9,5),(5,10,6),(5,10,8),(5,10,10),
    (9,10,10),(10,10,10)
(0,0,0),(0,0,0),(0,0,0),(6,3,7),(6,5,7),(7,7,7),(7,8,7),(8,8,7),(8,8,9),
    (8,8,9)
```

前三行是线程在进行更新，最后两行是线程仅进行读取。每个三元组是一组变量 x、y 和 z 以这样的顺序经历循环。这个输出中有几点要注意。

- 第一系列值显示了 x 在每个三元组里依次增加 1，第二系列 y 依次增加 1，以及第三系列 z 依次增加 1。
- 每个三元组的 x 元素仅在给定系列中自增，y 和 z 元素也一样，但增量不是平均的，而且在所有线程之间的相对顺序也不同。
- 线程 3 并没有看到任何的 x 或 y 的更新，它只能看到它对 z 的更新。然而这并不能阻止其他线程看到对 z 的更新混合在对 x 和 y 的更新中。

这对于松散操作是一个有效的结果，但并非唯一有效的结果。任何由三个变量组成，每个变量轮流保存从 0 到 10 的值，同时使得线程递增给定的变量，并打印出该变量从 0 到 9 的值的一系列值，都是有效的。

### 4. 理解松散顺序

为了理解这是如何工作的，想象每个变量是一个在小隔间里使用记事本的人。在他的记事本上有一列值。你可以打电话给他，要求他给你一个值，或者你可以告诉他写下了一个新值。如果你告诉他写下新值，他就将其写在列表底部。如果你向他要一个值，他就为你从列表中读取一个数字。

第一次你跟这个人交谈，如果你向他要一个值，此时他可能从他记事本上的列表里任意给你一个值。如果你接着向他要另一个值，他可能会再次给你同一个值，或是从列表下方给你一个。他永远不会给你一个在列表中更上面的值。如果你告诉他写下一个数字，然后再要一个值，他要么给你刚才你让他写下的数字，或者是列表上在那以下的数字。

假设某一次他的列表以这些值开始 5、10、23、3、1、2。如果你要一个值，你会得到其中的任意一个。如果他给你 10，下一次他可能再给你一个 10，或者后面的其他值，但不会是 5。如果你呼叫他 5 次，举个例子，他可能会说“10、10、1、2、2”。如果你告诉他写下 42，他会将其添加在列表末尾。如果你再向他要数字，他将一直告诉你“42”，直到他的列表上有另一个数，并且他愿意告诉你时。

现在，假设你的朋友 Carl 也有这个人的号码。Carl 也可以打电话给他，要么请他写下一个数字或是索取一个数字，他跟对待你一样，对 Carl 应用相同的规则。他只有一部电话，因此他一次只能处理你们中的一个人，所以他记事本上的列表是一个非常直观的列表。但是，仅仅因为你让他写下了新的号码，并不意味着他必须将其告诉 Carl，反之亦然。如果 Carl 向他索取一个数字，并被告知“23”，然后仅仅因为你要求这个人写下 42 并不意味着他下一次就会告诉 Carl。他可能会将 23、3、1、2、42 这些数字中的任何一个告诉 Carl，或者甚至是在你呼叫他之后，Fred 告诉他写下来的 67。他很可能会告诉 Carl“23、3、3、1、67”，这与他告诉你的

也没什么矛盾。就像是他为每个人设了一个小小的移动便签，把他对谁说了什么数都进行了记录，如图 5.5 所示。

现在假设不仅仅是一个人在一个小隔间，而是整个隔间群，有一大帮带着电话和笔记本的人。这些都是我们的原子变量。每一个变量都有自己的修改顺序（笔记本上的列表的值），但是它们之间完全没有关系。如果任意一个呼叫者（你、Carl、Anne、Dave 和 Fred）是一个线程，那么这就是每个操作都使用 memory_order_relaxed 时你所得到的东西。还有一些你可以告诉隔间里的人的额外的事情，比如“记下这个号码，并告诉我列表的底部是什么”（exchange）和“写下这个数字，如果列表底部的数字正是它，否则告诉我应该猜到什么”（compare_exchange_strong），但是这并不影响一般的原则。

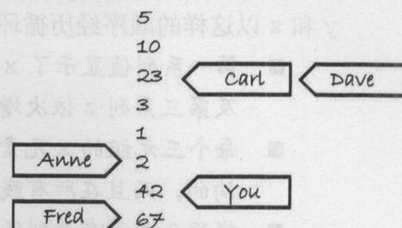


图 5.5 小隔间里的人的笔记本

如果你仔细地想一想清单 5.5 中的程序逻辑，write_x_then_y 就像是某个家伙打电话告诉小隔间 x 中的人让他写下 true，然后再打电话给小隔间 y 中的人让他写下 true。运行 read_y_then_x 的线程不断地呼叫小隔间 y 中的人询问一个值，一直到他说 true，然后再向小隔间 x 中的人询问值。这个在小隔间 x 中的人没有义务告诉你他列表上的任何一个具体的值，并且还有权利说 false。

这就使得松散的原子操作难以处理。他们必须与具有更强的顺序语义的原子操作结合起来使用，以便在线程间同步中发挥作用。我强烈建议避免松散的原子操作，除非绝对必要，即便这样，也应该极其谨慎地使用之。在清单 5.5 中，仅仅是两个线程和两个变量就能让所得到的结果这样不直观，鉴于此，不难想象在涉及更多线程和变量的时候，会变得多么复杂。

一种避免全面顺序一致性开销的达到额外同步的方法，是使用**获取-释放顺序**。

## 5. 获取-释放顺序

获取-释放顺序是松散顺序的进步，操作仍然没有总的顺序，但的确引入了一些同步。在这种顺序模型下，原子载入是**获取（aquire）**操作（memory_order_acquire），原子存储是**释放（release）**操作（memory_order_release），原子的读-修改-写操作（例如 fetch_add() 或 exchange()）是**获取**、**释放**或两者兼备（memory_order_acq_rel）。同步在进行释放的线程和进行获取的线程之间是对偶的。**释放操作与读取写入值的获取操作同步**。这意味着，不同的线程仍然可以看到不同的排序，但这些顺序是受到限制的。清单 5.7 采用获取-释放语义而不是顺序一致顺序，对清单 5.4 进行了改写。

**清单 5.7 获取-释放并不意味着总体排序**

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
    x.store(true,std::memory_order_release);
}

void write_y()
{
    y.store(true,std::memory_order_release);
}

void read_x_then_y()
{
    while(!x.load(std::memory_order_acquire));
    if(y.load(std::memory_order_acquire))      ←❶
        ++z;
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));
    if(x.load(std::memory_order_acquire))      ←❷
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);      ←❸
}
```

在这个例子里，断言 ❸ 可以触发（就像在松散顺序中的例子），因为对 x 的载入 ❷ 和对 y 的载入 ❶ 都读取 false 是可能的。x 和 y 由不同的线程写入，所以每种情况下从释放到获取的顺序对于另一个线程中的操作是没有影响的。

图 5.6 展示了清单 5.7 中的 happens-before 关系，伴随一种可能的结果，即两个读线

程看到了世界的不同方面。这可能是因为没有像前面提到的 happens-before 关系来强制顺序。

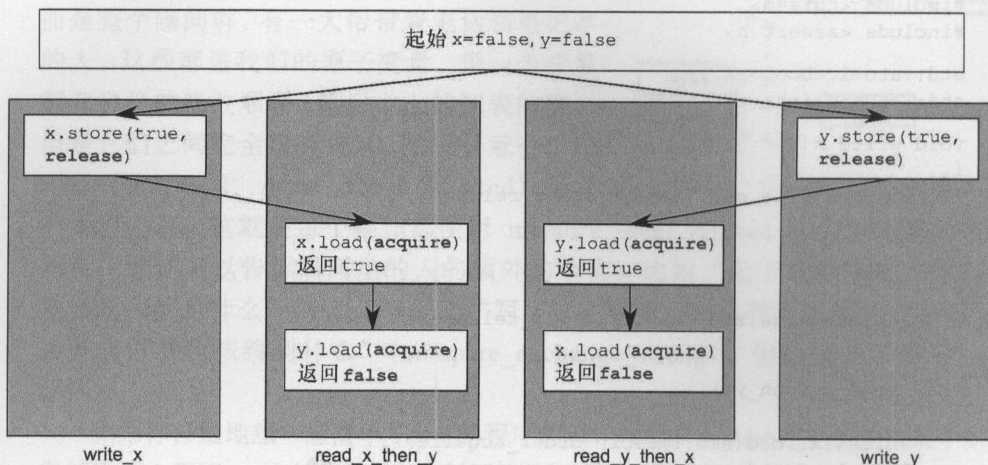



图 5.6 获取-释放和 happens-before

为看到获取-释放顺序的优点，你需要考虑类似清单 5.5 中来自同一个线程中的两个存储。如果你将对 y 的存储更改为使用 memory_order_release，并且让对 y 的载入使用类似于清单 5.8 中的 memory_order_acquire，那么你实际上就对 x 上的操作施加了一个顺序。

**清单 5.8 获取-释放操作可以在松散操作中施加顺序**

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);    <-❶ 旋转，等待 y 被设为 true
    y.store(true,std::memory_order_release);    <-❷
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));  <-❸
    if(x.load(std::memory_order_relaxed))       <-❹
        ++z;
}
```

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);          <-❺
}
```

最终，对 y 的加载❸将会看到由存储写下的 true❷。因为存储使用 memory_order_release 并且载入使用 memory_order_acquire，存储与载入同步。对 x 的存储❶发生在对 y 的存储❷之前，因为它们在同一个线程里。因为对 y 的存储与从 y 的载入同步，对 x 的存储同样发生于从 y 的载入之前，并且推而广之发生于从 x 的载入❹之前。于是，从 x 的载入必然读到 true，而且断言❺不会触发。如果从 y 的载入不在 while 循环里，就不一定是这个情况；从 y 的载入可能读到 false，在这种情况下，对从 x 读取到的值就没有要求了。为了提供同步，获取和释放操作必须配对。由释放操作存储的值必须被获取操作看到，以便两者中的任意一个生效。如果❷处的存储或❸处的载入有一个是松散操作，对 x 的访问就没有顺序，因此就不能确保❹处的载入读取 true，且 assert 会被触发。

你仍然可以用带着笔记本在他们小隔间的人们的形式来考虑获取-释放顺序，但是你必须对模型添加更多。首先，假定每个已完成的存储都是一批更新的一部分，因此当你呼叫一个人让他写下一个数字时，你也要告诉他这次更新是哪一批的一部分："请写下 99，作为第 423 批的一部分"。对于一批的最后一次存储，你也可以这样告诉他："请写下 147，作为第 423 批的最后一次的存储"。小隔间中的人会及时地写下这一信息，以及谁给了他这些值。这模拟了一个存储-释放操作。下一次，你告诉某人写下一个值，你应增加批次号码："请写下 41，作为第 424 批的一部分"。

当你询问一个值时，会有一个选择：你可以仅仅询问一个值（这是个松散载入），在此情况下这个人只会给你一个数字，或者你可以询问一个值以及它是不是这一批中最后一个数的信息（模拟载入-获取）。如果你询问批次信息，并且该值不是这一批中的最后一个，他会告诉你类似于"这个数字是 987，仅仅是一个'普通'的值"，然而如果它曾经是这一批中的最后一个，他会告诉你类似于"这个数字是 987，是来自 Anne 的第 956 批的最后一个数"。现在，获取-释放语义闪耀登场：如果当你询问一个值时告诉他你所知道的所有批次，他会从他的列表中向下查找，找到你所知道的任意一个批次的最后一个值，并且要么给你那个数字，要么列表更下方的一个数字。

这是如何模拟获取-释放语义的呢？让我们看一看这个例子。最开始，线程 a 运行 write_x_then_y 并对小隔间 x 内的人说，“请写下 true 作为线程 a 第一批的一部分”，然后他及时地写下了。线程 a 随后对小隔间 y 内的人说，“请写下 true 作为线程 a 第一批的最后一笔”，他也及时地写下了。与此同时，线程 b 运行着 read_y_then_x。线程 b 持续地向小隔间 y 里的人索取带着批次信息的值，直到他说“true”。他可能要询问很多次，但是最终这个人总会说“true”。但是在小隔间 y 里的人不能**仅仅**说“true”；他还要说“这是线程 a 第一批的最后一笔”。

现在，线程 b 继续向盒子 x 里的人询问值，但是这一次他说：“请让我拥有一个值，并且通过这一方式让我知晓线程 a 的第一批”。所以现在，小隔间 x 中的人必须查看他的列表，找到线程 a 中第一批的最后一次提及。他所具有的唯一一次提及是 true 值，同时也是列表上的最后一个值，所以他**必须**读取该值；否则，他会破坏游戏规则。

如果你回到 5.3.2 节查看**线程间 happens-before** 的定义，一个重要的内容就是它的传递性：如果 **A** 线程间发生于 **B** 之前，并且 **B** 线程间发生于 **C** 之前，则 **A** 线程间发生于 **C** 之前。这意味着获取-释放顺序可以用来在多个线程之间同步数据，甚至在“中间线程”并没有实际接触数据的场合。

## 6. 获取-释放顺序传递性同步

为了考虑传递性顺序，你需要至少三个线程。第一个线程修改一些共享变量，并对其中一个进行存储-释放。第二个线程接着对应于存储-释放，使用载入-获取来读取该变量，并且在第二个共享变量执行存储-释放。最后，第三个线程在第二个共享变量上进行载入-获取。在载入-获取操作看到存储-释放操作所写入的值以确保 synchronizes-with 关系的前提下，第三个线程可以读取由第一个线程存储的其他变量的值，即使中间线程没有动其中的任何一个。该情景展示于清单 5.9 中。

**清单 5.9　使用获取和释放顺序的传递性同步**

```
std::atomic<int> data[5];
std::atomic<bool> sync1(false),sync2(false);
void thread_1()
{
    data[0].store(42,std::memory_order_relaxed);
    data[1].store(97,std::memory_order_relaxed);
    data[2].store(17,std::memory_order_relaxed);
    data[3].store(-141,std::memory_order_relaxed);
    data[4].store(2003,std::memory_order_relaxed);    ❶ 设置 sync1
    sync1.store(true,std::memory_order_release);  <--
}
void thread_2()
{
```

```
    while(!sync1.load(std::memory_order_acquire));    ❷ 循环直到sync1被设置
    sync2.store(true,std::memory_order_release);      ❸ 设置sync2
}
void thread_3()
{
    while(!sync2.load(std::memory_order_acquire));    ❹ 循环直到sync2被设置
    assert(data[0].load(std::memory_order_relaxed)==42);
    assert(data[1].load(std::memory_order_relaxed)==97);
    assert(data[2].load(std::memory_order_relaxed)==17);
    assert(data[3].load(std::memory_order_relaxed)==-141);
    assert(data[4].load(std::memory_order_relaxed)==2003);
}
```

即使 thread_2 只动了变量 sync1❷和 sync2❸，也足以同步 thread_1 和 thread_3 以确保 assert 不会触发。首先，从 thread_1 存储 data 发生于对 sync1 的存储❶之前，因为他们在同一个线程里是 sequenced-before 关系。因为 sync1❶的载入在 while 循环中，它最终会看到 thread_1 中存储的值，并且因此形成释放-获取对的另一半。因此，对 sync1 的存储发生于 sync1 在 while 循环中的最后的载入之前。该载入的顺序在 sync2❸之前（因而也是 happens-before 的），与 thread_3 中 while 循环❹构成了一对释放-获取。对 sync2 的存储❸也就因而发生在载入❹之前，又发生在从 data 载入之前。因为 happens-before 的传递性质，你可以将它们串起来：对 data 的存储发生在对 sync1 的存储❶之前，又发生在从 sync1 的载入❷之前，又发生在对 sync2 的存储❸之前，又发生在从 sync2 的载入❹之前，又发生在 data 的载入之前。因此 thread_1 中对 data 的存储发生在 thread_3 中从 data 载入之前，并且 asserts 不会触发。

在这个例子中，你通过使用在 thread_2 中的一个具有 memory_order_acq_rel 的读-修改-写操作，将 sync1 和 sync2 合并为一个变量。有一个选项是使用 compare_exchange_strong()，以确保该值只有当 thread_1 中的存储可见时才能被更新。

```
std::atomic<int> sync(0);
void thread_1()
{
    // ...
    sync.store(1,std::memory_order_release);
}
void thread_2()
{
    int expected=1;
    while(!sync.compare_exchange_strong(expected,2,
                                        std::memory_order_acq_rel))
        expected=1;
}
void thread_3()
{
```

```
    while(sync.load(std::memory_order_acquire)<2);
    // ...
}
```

如果你使用的读-修改-写操作，挑选你所希望的语义是很重要的。在这种情况下，你同时需要获取和释放语义，因此 memory_order_acq_rel 是合适的，但是你也可以使用其他的顺序。具有 memory_order_acquire 语义的 fetch_sub 操作不与任何东西同步，即便是它存储一个值，因为它并不是一个释放操作。同样地，一次存储无法与具有 memory_order_release 语义的 fetch_or 操作同步，因为 fetch_or 的读取部分并不是一个获取操作。具有 memory_order_acq_rel 语义的读-修改-写操作表现得既像获取又像释放，所以之前的一次存储可以与这样的操作同步，并且它也可以与之后的一次载入同步，就像这个例子中的情况一样。

如果你将获取-释放操作与顺序一致操作混合起来，顺序一致载入表现的像是获取语义的载入，并且顺序一致存储表现的像是释放语义的存储。顺序一致的读-修改-写操作表现得既像获取又像释放操作。松散操作仍然是松散的，但是会收到额外的 synchronizes-with 和随之而来的 happens-before 关系的约束，这是由于获取-释放语义的使用而引入的。

暂且不管可能存在的不直观的后果，任何使用锁的人都不得不面对相同的顺序问题，锁定互斥元是一个获取操作，而解锁互斥元是一个释放操作。对于互斥元，你了解到必须确保当你读取一个值的时候，锁定与你写它时曾锁定的相同的互斥元；你的获取和释放操作必须是对同一个变量以确保顺序。如果数据受到互斥元的保护，锁的独占特性意味着结果与令它的锁定与解锁成为顺序一致操作所得到结果没有区别。类似地，如果你在原子变量上使用获取与释放顺序建立一个简单的锁，那么从使用该锁的代码的角度来看，其行为会表现为顺序一致，即便内部的操作并非这样。

如果你对你的原子操作不需要顺序一致顺序的严格性，获取-释放顺序的配对同步可能会比顺序一致操作所要求的全局顺序有着低得多的同步成本。这里需要权衡的是确保该顺序能够正确工作，以及线程间不直观的行为不会出问题所需求的脑力成本。

### 7. 使用获取-释放顺序和 MEMORY_ORDER_CONSUME 的数据依赖

在本节的绪论中我提到了 memory_order_consume 是获取-释放顺序模型的一部分，但前面的描述中它很明显的缺失了。这是因为 memory_order_consume 是很特别的，它全是关于数据依赖，它引入了与 5.3.2 节中提到的线程间 happens-before 关系有着细微差别的数据依赖。

有两个处理数据依赖的新的关系：**依赖顺序在其之前（dependency-ordered-before）**和**带有对其的依赖（carries-a-dependency-to）**。与 sequenced-before 相似，carries-a-

dependency-to 严格适用于单个线程之内，是操作间数据依赖的基本模型。如果操作 A 的结果被用作操作 B 的操作数，那么 A 带有对 B 的依赖。如果操作 A 的结果是类似 int 的标量类型的值，那么如果 A 的结果存储在一个变量中，并且该变量随后被用作操作 B 的操作数，此关系也是适用的。这种操作也具有传递性，所以如果 A 带有对 B 的依赖且 B 带有对 C 的依赖，那么 A 带有对 C 的依赖。

另一方面，dependency-ordered-before 的关系可以适用于线程之间。它是通过使用标记了 memory_order_consume 的原子载入操作引入的。这是 memory_order_acquire 的一种特例，它限制了对直接依赖的数据同步。标记为 memory_order_release、memory_order_acq_rel 或 memory_order_seq_cst 的存储操作 A 的依赖顺序在标记为 memory_order_consume 的载入操作之前，如果此次消耗读取所存储值的话。如果载入使用 memory_order_acquire，那么这与 synchronizes-with 关系所得到的是相反的。如果操作 B 带有对操作 C 的某种依赖，那么 A 也是依赖顺序在 C 之前。

如果这对线程间 happens-before 关系没有影响，那么在同步目的上就无法为你带来任何好处，但它的确实现了：如果 A 依赖顺序在 B 之前，则 A 也是线程间发生于 B 之前。

这种内存顺序的一个重要用途，是在原子操作载入指向某数据的指针的场合。通过在载入上使用 memory_order_consume 以及在之前的存储上使用 memory_order_release，你可以确保所指向的数据得到正确地同步，无需在其他非依赖的数据上强加任何同步需求。清单 5.10 显示了这种情况的例子。

**清单 5.10 使用 std::memory_order_consume 同步数据**

```
struct X
{
    int i;
    std::string s;
};

std::atomic<X*> p;
std::atomic<int> a;

void create_x()
{
    X* x=new X;
    x->i=42;
    x->s="hello";                                      ❶
    a.store(99,std::memory_order_relaxed);          <-┘
    p.store(x,std::memory_order_release);       <-❷
}

void use_x()
{
```

```
    X* x;
    while(!(x=p.load(std::memory_order_consume)))      ❸
        std::this_thread::sleep(std::chrono::microseconds(1));
    assert(x->i==42);                                   ❹
    assert(x->s=="hello");                              ❺
    assert(a.load(std::memory_order_relaxed)==99);      ❻
}

int main()
{
    std::thread t1(create_x);
    std::thread t2(use_x);
    t1.join();
    t2.join();
}
```

即使对 a 的存储❶的顺序在对 p 的存储❷之前，并且对 p 的存储被标记为 memory_order_release，p 的载入❸被标记 memory_order_consume。这意味着，对 p 的存储只发生在依赖 p 的载入值的表达式之前。这还意味着，在 X 结构的数据成员❹、❺上的断言保证不会被触发，因为 p 的载入带有对那些通过变量 x 的表达式的依赖。另一方面，在 a 的值上的断言❻或许会触发，或许不被触发；此操作并不依赖于从 p 载入的值，因而对读到的值就没有保证。如你所见，因为它被标记为 memory_order_relaxed，所以特别的显著。

有时候，你并不希望四处带着依赖所造成的开销。你想让编译器能够在寄存器中缓存值，并且对操作进行重新排序以优化代码，而不用担心依赖。在这些情形下，你可以使用 std::kill_dependency()显式地打破依赖链条。std::kill dependency()是一个简单的函数模板，它将所提供的参数复制到返回值，但这样做会打破依赖链条。例如，如果你有一个全局的的只读数组，你在从另外的线程中获取该数组的索引时使用了 std::memory order consume，你可以用 std::kill_dependency()让编译器知道它无需重新读取数组项的内容，就像下面的这个例子。

```
int global_data[]={ … };
std::atomic<int> index;
void f()
{
    int i=index.load(std::memory_order_consume);
    do_something_with(global_data[std::kill_dependency(i)]);
}
```

当然，在这样一个简单的场景里你一般根本不会使用 std::memory_order_consume，但是你可能会在具有更复杂代码的相似情形下调用 std::kill_dependency()。你必须记住这是一项优化，所以只应小心地使用，用在优化器表明需要使用的地方。

现在，我已经涵盖了内存顺序的基础，是时候看看 synchronizes-with 关系中更复杂的部分，即体现为**释放序列（release sequence）**的形式。

## 5.3.4 释放序列和 synchronizes-with

早在 5.3.1 节，我提到过你可以在对原子变量的载入，和来自另一个线程的对该原子变量的载入之间，建立一个 synchronizes-with 关系，即便是在存储和载入之间有一个读-修改-写顺序的操作存在的情况下，前提是所有的操作都作了适当的标记。现在既然我已经介绍了可能的内存顺序“标签”，我就可以详细解释这一点。如果存储被标记为 memory_order_release、memory_order_acq_rel 或 memory_order_seq_cst，载入被标记为 memory_order_consume、memory_order_acquire 或 memory_order_seq_cst，并且链条中的每个操作都载入由之前操作写入的值，那么，该操作链条就构成了一个**释放序列**（**release sequence**），并且最初的存储与最终的载入是 synchronizes-with 的（例如 memory_order_acquire 或 memory_order_seq_cst）或者是 dependency-ordered-before 的（例如 memory_order_consume）。该链条中的所有原子的读-修改-写操作都可以具有任意的内存顺序（甚至是 memory_order_relaxed）。

要了解这是什么意思以及它为什么很重要，考虑 atomic<int>作为在一个共享的队列中的项目数量的 count，如清单 5.11 所示。

**清单 5.11 使用原子操作从队列中读取值**

```
#include <atomic>
#include <thread>

std::vector<int> queue_data;
std::atomic<int> count;

void populate_queue()
{
    unsigned const number_of_items=20;
    queue_data.clear();
    for(unsigned i=0;i<number_of_items;++i)
    {
        queue_data.push_back(i);
    }
                                                                  ❶ 最初的
    count.store(number_of_items,std::memory_order_release);      <-  存储
}

void consume_queue_items()                                  一个读-修改-   ❷
{                                                           写操作
    while(true)
    {
        int item_index;
        if((item_index=count.fetch_sub(1,std::memory_order_acquire))<=0)   <-
        {
            wait_for_more_items();   <- 等待更多
            continue;             ❸ 项目
```

```
        }
        process(queue_data[item_index-1]);   <-- ❹ 读取 queue_data 是安全的
    }
}

int main()
{
    std::thread a(populate_queue);
    std::thread b(consume_queue_items);
    std::thread c(consume_queue_items);
    a.join();
    b.join();
    c.join();
}
```

处理事情的方法之一是让产生数据的线程将项目存储在一个共享的缓冲区中，然后执行 count.store(number_of_items,memory_order_release)❶让其他线程知道数据可用。消耗队列项目的线程接着可能会执行 count.fetch_sub(1, memory_order_acquire)❷队列中索取一个项目，在实际读取共享缓冲区❹之前。一旦 count 变为零，又没有更多的项目，线程就必须等待❸。

如果只有一个消费者线程，这是良好的。fetch_sub()是一个具有 memory_order_acquire 语义的读取，并且存储具有 memory_order_release 语义，所以存储与载入同步，并且该线程可以从缓冲区里读取项目。如果有两个线程在读，第二个 fetch_sub()将会看到由第一个写下的值，而非由 store 写下的值。若没有该释放序列的规则，第二个线程不会具有与第一个线程的 happens-before 关系，并且读取共享缓冲区也不是安全的，除非第一个 fetch_sub()也具有 memory_order_release 语义，这会带来两个消费者线程之间不必要的同步。如果在 fetch_sub 操作上没有释放序列规则或是 memory_order_release，就没有什么能要求对 queue_data 的存储对第二个消费者可见，你就会遇到数据竞争。幸运的是，第一个 fetch_sub()的确参与了释放序列，并因此 store()与第二个 fetch_sub()同步。这两个消费者线程之间依然没有 synchronizes-with 关系。这如图 5.7 所示。图 5.7 中的虚线表示的是释放序列，实线表示的是 happens-before 关系。

在这一链条中，可以有任意数量的链接，但前提是它们都是类似 fetch_sub()这样的读-修改-写操作，store()仍然与每一个具有 memory_order_acquire 标记的操作同步。在这个例子里，所有的链接都是一样的，都是获取操作，但它们可以由具有不同的内存顺序语义的不同操作组合而成。

尽管大多数的同步关系来自于应用到原子变量操作的内存顺序语义，但通过屏障（**fence**）来引入额外的顺序约束也是可能的。

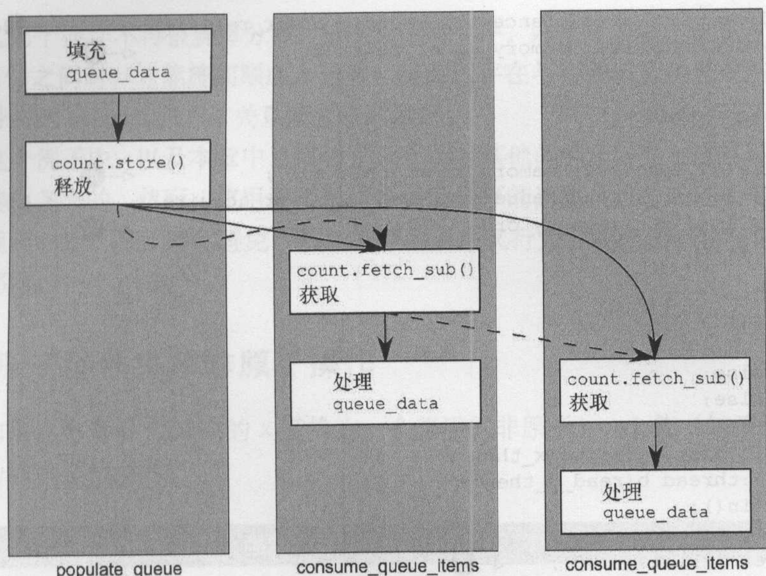



图 5.7 清单 5.11 中队列操作的释放序列

## 5.3.5 屏障

没有一套屏障的原子操作库是不完整的。这些操作可以强制内存顺序约束，而无需修改任何数据，并且与使用 memory_order_relaxed 顺序约束的原子操作组合起来使用。屏障是全局操作，能在执行该屏障的线程里影响其他原子操作的顺序。屏障一般也被称为内存障碍（**memory barriers**），它们之所以这样命名，是因为它们在代码中放置了一行代码，使得特定的操作无法穿越。也许你还记得 5.3.3 节中，在独立变量上的松散操作通常可以自由地被编译器或硬件重新排序。屏障限制了这一自由，并且在之前并不存在的地方引入 happens-before 和 synchronizes-with 关系。

让我们从在清单 5.5 中的每个线程上的两个原子操作之间添加屏障开始，如清单 5.12 所示。

**清单 5.12 松散操作可以使用屏障来排序**

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);     <-❶
```

```
    std::atomic_thread_fence(std::memory_order_release);     ❷
    y.store(true,std::memory_order_relaxed);               ❸
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));             ❹
    std::atomic_thread_fence(std::memory_order_acquire);     ❺
    if(x.load(std::memory_order_relaxed))                  ❻
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);     ❼
}
```

释放屏障❷与获取屏障❺同步，因为从 y 的载入❹在读取存储在❸的值。这意味着对 x 的存储❶发生在从 x 的载入❻之前，所以读取的值必然是 ture，并且在❼处的断言不会触发。这与原来没有屏障的情况下，对 x 的存储和从 x 的载入没有顺序，是相反的，所以断言可以触发。注意，这两个屏障都是必要的，你需要在一个线程中有一个释放，在另一个线程中有一个获取，从而实现 synchronizes-with 关系。

在这种情况下，释放屏障❷的效果，与对 y 的存储❸被标记为 memory_order_release 而不是 memory_order_relaxed 是相似的。同样，获取屏障❺令其与从 y 的载入❹被标记为 memory_order_acquire 相似。这是屏障的总体思路：如果获取操作看到了释放屏障后发生的存储的结果，该屏障与获取操作同步；如果在获取屏障之前发生的载入看到释放操作的结果，该释放操作与获取屏障同步。当然，你可以在两边都设置屏障，就像在这里的例子一样，在这种情况下，如果在获取屏障之前发生的载入看见了释放屏障之后发生的存储所写下的值，该释放屏障与获取屏障同步。

尽管屏障同步依赖于在屏障之前或之后的操作所读取或写入的值，注意同步点就是屏障本身是很重要的。如果你从清单 5.12 中将 write_x_then_y 拿走，将对 x 的写入移到下面的屏障之后，断言中的条件就不再保证是真的，即便对 x 的写入在对 y 的写入之前。

```
void write_x_then_y()
{
    std::atomic_thread_fence(std::memory_order_release);
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_relaxed);
}
```

这两个操作不再被屏障分隔，所以也不再有序。只有当屏障出现在对 x 的存储和对 y 的存储之间时，才能施加顺序。当然，屏障的存在与否并不影响现存的，由其他原子操作带来的 happens-before 关系所强制的顺序。

这个例子中，以及本章中目前为止几乎所有其他的例子，完全是从具有原子类型的变量建立起来的。然而，使用原子操作来强制顺序的真正优点，是它们可以在非原子操作上强制顺序，并且因此避免了数据竞争的未定义行为，就像之前你在清单 5.2 中所看到的那样。

## 5.3.6　用原子操作排序非原子操作

如果你将清单 5.12 中的 x 替换为一个普通的非原子 bool 值（如清单 5.13 所示），该行为确保是相同的。

清单 5.13　在非原子操作上强制顺序

```
#include <atomic>
#include <thread>
#include <assert.h>

bool x=false;                          // x现在是一个普通的非原子变量
std::atomic<bool> y;
std::atomic<int> z;

void write_x_then_y()
{
    x=true;                                          // ① 在屏障前存储x
    std::atomic_thread_fence(std::memory_order_release);
    y.store(true,std::memory_order_relaxed);         // ② 在屏障后存储y
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));       // ③ 等待到你看见来自#2 的写入
    std::atomic_thread_fence(std::memory_order_acquire);
    if(x)                                            // ④ 将读取#1 写入的值
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);                             // ⑤ 此断言不会触发
}
```

屏障仍然为对 x 的存储 ❶、对 y 的存储 ❷、从 y 的载入 ❸ 和从 x 的载入 ❹ 提供了强制顺序，并且在对 x 的存储和从 x 的载入之间仍然有 happens-before 关系，所以断言 ❺ 还是不会触发。存储 ❷ 和载入 ❸y 仍然必须是原子的；否则，在 y 上就会有数据竞争，但是屏障在 x 的操作上强制了顺序，一旦读线程看见了已存储的 y 值。这个强制顺序意味着 x 上没有数据竞争，即使它被一个线程修改并且被另一个线程读取。

并不是只有屏障才能排序非原子操作。你之前在清单 5.10 中看到的用 `memory_order_release/memory_order_consume` 对偶来排序对动态分配对象的访问，以及本章中的许多例子，都可以通过将其中一些 `memory_order_relaxed` 操作替换为普通的非原子操作来进行重写。

通过使用原子操作来排序非原子操作，是 happens-before 中的 sequenced-before 部分变得如此重要的所在。如果一个非原子操作的顺序在一个原子操作之前，同时该原子操作发生于另一个线程中的操作之前，这个非原子操作同样发生在另一个线程的该操作之前。这就是清单 5.13 中 x 上的操作顺序的来历，也是清单 5.2 中的示例可以工作的原因。这也是 C++标准库中更高级别的同步功能的基础，比如互斥元和条件变量。想看一看这是如何工作的，考虑一下清单 5.1 中的简单的自旋锁互斥元。

`lock()` 操作是在 `flag.test_and_set()` 上的循环，使用的是 `std::memory_order_acquire` 顺序，而 `unlock()` 是对 `flag.clear()` 的调用，用的是 `std::memory_order_release` 顺序。当第一个线程调用 `lock()` 时，标志被初始化为清除，所以第一次调用 `test_and_set()` 将设置标志并返回 `false`，表明该线程现在拥有了锁，并且终止循环。线程接下来就可以自由地修改被互斥元保护的任何数据。此时所有其他调用 `lock()` 的线程会发现标志已经被设置，而且会被阻塞在 `test_and_set()` 循环中。

当持有锁的线程已完成修改受保护的数据时，它会调用 `unlock()`，继而调用具有 `std::memory_order_release` 语义的 `flag.clear()`。然后它会与后续的来自另一线程上 `lock()` 的调用中的对 `flag.test_and_set()` 的调用同步，因为该调用具有 `std::memory_order_acquire` 语义。由于对受保护数据的修改必然顺序在 `unlock()` 之前，于是也就发生在后续的来自于第二个线程的 `lock()` 之前（因为 `unlock()` 和 `lock()` 之间的 synchronizes-with 关系），并且发生在来自于第二个线程的对数据的任意访问之前，一旦第二个线程获取了锁。

虽然其他的互斥元的实现会具有不同的内部操作，但其基本原理是相同的，`lock()` 是在一个内部内存地址上的获取操作，`unlock()` 是在相同内存地址上的释放操作。

## 5.4 小结

本章介绍了 C++11 内存模型的底层细节，以及在线程间提供同步基础的原子操作。

这包括了由 std::atomic<>类模板的特化提供的基本原子类型，由 std::atomic<>主模板提供的泛型原子接口，在这些类型上的操作，以及各种内存顺序选项的复杂细节。

我们还看了屏障，以及它们如何通过原子类型上的操作配对，以强制顺序。最后，我们回到开头，看了看原子操作是如何用来在独立线程上的非原子操作之间强制顺序的。

在接下来的章节中，我们将学习在原子操作之外使用高级同步功能，来为并发访问设计高效的容器，而且我们会编写并行处理数据的算法。

# 第6章　设计基于锁的并发数据结构

本章主要内容

- 为并发设计数据结构的含义
- 这么做的准则
- 实现设计满足并发性的数据结构的例子

上一章中我们寻找原子操作和存储器模式的底层细节。在这一章中，我们先不探讨底层细节（尽管第7章中我们需要用到它们）而是考虑数据结构。

程序设计问题中选择什么样的数据结构是总体解决方法的关键部分，对于并行程序设计问题也不例外。如果多线程用到了数据结构，那么此数据结构要么完全不可变，即从不发生变化也不需要同步，要么程序设计中就要保障线程间能正确同步变化。一种选择就是使用单独的互斥元和外部锁来保护数据，使用我们在第3章和第4章中提到的技术，另外一种选择就是设计一个可以同时访问的数据结构。

当为并发性设计数据结构时，你可以使用前面章节中介绍的多线程应用程序中的基本构造模块，例如互斥元和条件变量。实际上，我们已经看到了几个例子，显示了如何将这些基本部分组合起来写入数据结构并保证当前多线程数据的安全性。

这章中，我们首先考虑为并发性设计数据结构时的一般准则。然后我们采取基本构造模块和条件变量，并且在我们进入到更加复杂的数据结构前先回顾一下这些基础数据结构。在第7章中，我们考虑如何回到基础并且使用第5章中描述的原子操作来建立无锁的数据结构。

那么，言归正传，让我们考虑为并发性设计一种数据结构时需要涉及到哪些方面。

# 6.1 为并发设计的含义是什么

在最基本的层面，为并发设计数据结构意味着多个线程可以同时使用此数据结构，执行相同或不同的操作，并且每个线程都有数据结构的一致性视图。不会丢失或破坏数据，维持所有不变量，并且没有不确定的竞争条件，此种数据结构就被称为线程安全的。通常，只有在特定的并发存取下，一种数据结构才是安全的。很有可能出现这种情况，就是多个线程对数据结构执行同一种操作，然而另一个线程的操作需要进行独占访问。或者，也许多个线程执行不同的操作，它们并发地存取某个数据结构是安全的。然而多个线程执行相同的操作，它们并发地存取某个数据结构可能会有问题。

实际上并发设计远远不只是为多个线程提供存取数据结构的**并发机会**。本质上，互斥元提供的是**互斥**，一次只允许一个线程获取互斥元的锁。一个互斥元通过明确阻止对它所保护的数据进行并发存取来保护数据结构。

这被称为**序列化**（**serialization**）：多个线程轮流存取互斥元保护的数据，它们必须线性地而非并发地存取数据。所以，你必须仔细考虑数据结构的设计来实现真正的并发存取。一些数据结构比别的数据结构在并发性上有更大的范围，但是在所有的情况下，其思想是一致的：更小的保护区域，更少的操作被序列化，以及更高的并发潜能。

在我们考虑某个数据结构设计前，我们先来回顾设计并发性时需要考虑的一些简单准则。

## 为并发设计数据结构的准则

就像我提到的，为并发存取设计数据结构时，你需要考虑两方面：保证存取是安全的以及允许真正的并发存取。第 3 章中我阐述了如何使得数据结构线程安全的基本原理。

- 保证当数据结构不变性被别的线程破坏时的状态不被任何别的线程看到。
- 注意避免数据结构接口所固有的竞争现象，通过为完整操作提供函数，而不是提供操作步骤。
- 注意当出现例外时，数据结构是怎样来保证不变性不被破坏的。
- 当使用数据结构时，通过限制锁的范围和避免使用嵌套锁，来降低产生死锁的机会。

在考虑这些细节前，先考虑使用数据结构时的限制条件也是很重要的，如果一个函数通过特殊函数使用数据结构，那么其他线程调用哪个函数是安全的？

这是要考虑的关键性问题。大多数构造函数和析构函数需要以独占方式访问数据结构，但是需要使用者保证它们在构造函数完成前或者析构函数开始后没有被使用。如果数据结构支持赋值、`swap()`、或复制构造，那么作为数据结构的设计者，就需要决定

这些操作与别的操作同时被调用时是否安全，或者是否需要使用者保证互斥访问，即使操作数据结构的大部分函数可能被多线程同时访问时没有问题。

第二个要考虑的方面就是实现真正的并发存取。我没办法在这里给出详尽的准则；而是，这里有一列问题，作为数据结构设计者需要问问你自己。

- 锁的范围能否被限定，使得一个操作的一部分可以在锁外被执行？
- 数据结构的不同部分能否被不同的互斥元保护？
- 是否所有操作需要同样级别的保护？
- 数据结构的一个小改变能否在不影响操作语义情况下提高并发性的机会？

所有这些问题都被一个想法所指导：如何能够最小化必然发生的序列化，并且能够最大限度地实现并发性？通常，当多个线程仅仅读取数据结构时可以并发访问此数据结构，但是一个线程必须以独占方式修改数据结构。使用构造函数 boost::shared_mutex 可以实现此功能。而且，很快你就会看到，数据结构支持执行不同操作的线程和执行相同操作的序列化线程并发访问它。

最简单的线程安全数据结构通常使用互斥元和锁来保护数据。就像第 3 章中提到的，比较简单的方式是保证每次只有一个线程使用此数据结构，尽管这种方式也会存在问题。为了让你轻松了解线程安全数据结构的设计，本章我们研究这种基于锁的数据结构，在第 7 章中我们将研究无锁的并发数据结构的设计。

## 6.2　基于锁的并发数据结构

设计基于锁的并发数据结构关键是要确保存取数据时要锁住正确的互斥元，并且要确保将锁的时间最小化。只用一个互斥元保护一个数据结构是很困难的。你需要确保此数据在互斥锁保护区域之外不会被存取，并且不会发生接口所固有的竞争现象。如果使用独立的互斥元来保护数据结构的独立部分，问题会变得更复杂，并且如果数据结构上的操作需要多个互斥元被锁住就有可能产生死锁。因此，设计有多个互斥元的数据结构时，需要比设计只有一个互斥元的数据结构考虑得更细致。

在这部分，将 6.1.1 节中的准则运用到一些简单数据结构的设计中，使用互斥元和锁来保护数据。在不同的情况下，你可以找到机会在确保数据结构仍然线程安全的情况下能够有更大的并发性。

首先我们回顾一下第 3 章中提到的栈实现，这大约是最简单的一种数据结构，而且它只是使用了一个互斥元。它是否真的线程安全？它距离实现真正的并发性到底有多远呢？

### 6.2.1　使用锁的线程安全栈

在清单 6.1 中会重现第 3 章中的线程安全栈。目的是为 std::stack<>写一个相似

的线程安全数据结构，支持数据入栈和出栈。

**清单 6.1 线程安全栈的类定义**

```
#include <exception>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));          ①
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();     ②
        std::shared_ptr<T> const res(
            std::make_shared<T>(std::move(data.top())));   ③
        data.pop();     ④
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=std::move(data.top());     ⑤
        data.pop();     ⑥
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

让我们轮流看看每个准则，以及它们是如何应用的。

首先，如你所见，基本的线程安全是通过使用互斥锁 m 保护成员函数来实现的。这就确保了同一时间只有一个线程在存取数据，因此每个成员函数都保持了不变量，没有线程会看到一个破坏的不变量。

其次，empty()和任何一个 pop()函数间都可能产生竞争条件，但是当 pop()持有锁的时候，这段代码会明确地检查它所包含的栈，因此竞争条件就不成问题了。调用 pop()时会直接返回出栈的数据项，就避免了类似于 std::stack<>中单独的成员函数 top()和 pop()间可能产生的竞争条件。

下一个，这里可能会抛出一些异常。锁住一个互斥元可能会抛出异常，这不仅是极其罕见的（因为这就表明互斥元有问题或者缺乏系统资源），也是每个成员函数的第一个操作。因为没有数据被修改，所以是安全的。解锁一个互斥元总是成功的，所以总是安全的，并且使用 std::lock_guard<>保证了在成员函数结束的时候互斥元不会被锁定。

调用 data.push()❶ 可能会抛出异常，如果复制或者移动数据项抛出异常或者不能分配足够的内存来扩展下层的数据结构。不管怎样，std::stack<>保证了它是安全的，因此这也不是一个问题。

在函数 pop()重载的第一种形式中，它的代码可能会抛出一个 empty_stack 异常 ❷，但是没有做任何修改，因此它是安全的。创建 res❸ 时因为一些原因可能会抛出异常：调用 std::make_shared 可能会抛出异常，因为它无法为新对象和需要引用计数的内部数据分配内存。当复制构造函数或移动构造函数中返回的数据项被复制/移动到新分配的内存时也可能会抛出异常。在这两种情况下，C++运行库和标准库确保没有内存泄露并且新对象（如果存在的话）被正确销毁。因为你仍然没有修改下层的栈，所以没有问题。作为结果的返回值，调用 data.pop()❹ 保证了不会抛出异常，因此 pop()的重载是异常安全的。

函数 pop()重载的第二种形式是类似的，只不过这次是拷贝赋值或移动赋值操作符抛出异常 ❺，而不是构造新对象和一个 std::shared_ptr 实例抛出异常。同样，你实际上并没有修改数据结构直到调用 data.pop()❻，这仍然保证了不会抛出异常，因此这一重载也是异常安全的。

最后，empty()不修改任何数据，因此是异常安全的。

这里会有产生死锁的可能，因为当持有锁时调用了用户代码：拷贝构造函数或移动构造函数 ❶、❸，以及内含数据项的拷贝赋值操作或移动赋值操作 ❺，以及可能由用户定义的 new 操作符。如果当数据项被插入栈或从栈中移出时，这些函数调用了栈的成员函数，或者当栈成员函数被调用时，这些函数请求一个锁的时候保持着另一个锁，就有可能产生死锁。然而，要求栈的使用者做出如下保证是明智的：你不能理所当然地在没有复制数据项或为之分配内存的情况下，将它加入栈或者从栈中移走它。

因为所有的成员函数都使用 std::lock_guard<>来保护数据，所以多个线程调用 stack 的成员函数是安全的。成员函数中只有构造函数和析构函数是不安全的，但是这不是一个特殊问题，对象只能被构造和销毁一次。在一个没有完全构造好的对象或部分析构对象上调用成员函数永远都不是一个好主意。因此，使用者必须保证在它被完全构造前别的线程不能存取栈，并且在它被完全销毁前，所有线程结束存取栈。

尽管对多线程来说，因为使用了锁，每次只有一个线程对栈数据结构进行操作，所以同时调用成员函数是安全的。但是当 stack 上存在显著的竞争时，线程**序列化**可能会限制应用的性能。当一个线程在等待锁的时候，它就做不了任何有用的工作。并且，栈没有为等待数据项被插入的线程提供任何方式的准备，因此如果一个线程需要等待，它就会反复地调用 empty()，或者只是调用 pop()，并且捕捉 empty_stack 异常。如果这种情况发生的话，这种栈实现就成为一个比较糟糕的选择，因为一个等待中的线程要么消耗宝贵的资源在检查数据上，要么用户必须写外部的等待和通知代码（比如，使用条件变量），而这就有可能让内部锁变得无效并且造成浪费。第 4 章中的队列在数据结构中使用了条件变量，这就提供了一种数据结构中包含等待的方法。因此下面我们来看一看。

## 6.2.2 使用锁和条件变量的线程安全队列

清单 6.2 中重现了第 4 章中提到的线程安全队列。类似于栈是以 std::stack<>建模的，队列是以 std::queue<>来建模的。此外，它的接口与标准容器适配器的接口是不同的，因为它要满足其数据结构对于来自多线程的并发访问是安全的这一约束。

**清单 6.2 使用条件变量的线程安全队列的完整类定义**

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(data));
        data_cond.notify_one();                    <-❶
    }
```

```
    void wait_and_pop(T& value)    <-❷
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=std::move(data_queue.front());
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()    <-❸
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});    <-❹
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();    <-❺
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

清单 6.2 中所示的队列实现的结构与清单 6.1 中所示的栈的结构是类似的，除了 push()❶ 中调用的 data_cond.notify_one() 以及 wait_and_pop() 函数❷、❸。try_pop() 的两种重载形式与清单 6.1 中的 pop() 函数的两种重载形式基本上是相同的，区别在于当队列为空的时候，try_pop() 函数不引发异常。try_pop() 函数要么返回一个表明是否取回一个值的 bool 值，要么在返回指针的重载❺没能取到值的时候返回 NULL，这也是实现栈的有效方式。因此，如果不包括 wait_and_pop() 函数，为

栈应用所做的分析同样适用此。

新的 wait_and_pop()函数是一种解决等待队列入口问题的方法，与反复调用 empty()函数不同，等待中的线程可以通过调用 wait_and_pop()函数，然后数据结构使用条件变量来处理这种等待状态。对 data_cond.wait()的调用直到下层队列中至少有一个元素时，才会返回，因此你不用担心在代码的这个地方队列为空的可能性，数据仍然被互斥元上的锁保护着。这些函数不会增加任何新的竞争条件和死锁的可能性，并且维持了不变量。

当一个元素进入队列时，如果不止一个线程处于等待状态，关于异常安全就会有点崎岖，只有一个线程会被 data_cond.notify_one()的调用唤醒。然而，如果被唤醒的线程在 wait_and_pop()中引发异常，例如构造 std::shared_ptr<>的时候❹，那么就没有线程将被唤醒。如果不能接受这一点，那么可以用 data_cond.notify_all()来代替 data_cond.notify_one()，前者将唤醒所有在等待的线程，代价就是当最后它们发现队列为空的时候，其中的大部分线程都要重新进入睡眠状态。第二种替代方法就是，当引发异常的时候，让 wait_and_pop()调用 notify_one()，这样另外一个线程可以尝试获取所存储的值。第三种替代方法，是将 std::shared_ptr<>的初始化移动到 push()调用，并且存储 std::shared_ptr<>实例而不是直接存储值。将内部的 std::queue<>复制到 std::shared_ptr<>并不会引发异常，因此 wait_and_pop()又是安全的了。清单 6.3 展示了使用这一思想重新修订以后的队列实现。

**清单 6.3　包含 std::shared_ptr<>实例的线程安全队列**

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<std::shared_ptr<T> > data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=std::move(*data_queue.front());   ❶
        data_queue.pop();
    }
    bool try_pop(T& value)
    {
```

```
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=std::move(*data_queue.front());     <-❷
        data_queue.pop();
        return true;
    }
    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res=data_queue.front();  <-┐
        data_queue.pop();                             ❸
        return res;
    }
    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res=data_queue.front();  <-❹
        data_queue.pop();
        return res;
    }
    void push(T new_value)
    {
        std::shared_ptr<T> data(
            std::make_shared<T>(std::move(new_value)));    <-❺
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

通过 std::shared_ptr<>持有数据的基本效果是直观的：接受一个对变量的引用来获取新值的 pop 函数，现在必须得解引用所存储的指针 ❶、❷，返回一个 std::shared_ptr<>实例的 pop 函数可以在返回调用前从队列中取得这个数 ❸、❹。

如果数据由 std::shared_ptr<>持有，那么有一个额外的好处：可以在 push() 的锁外面完成此新实例的分配 ❺，然而在清单 6.2 中，就必须在 pop() 持有锁的情况下才能这样做。因为内存分配通常是很昂贵的操作，这种方式就有助于提高队列的性能，因为它也减少了持有互斥元的时间，允许其他线程同时在队列上执行操作。

就像之前栈的例子一样，使用互斥元来保护整个数据结构限制了队列所支持的并

发。尽管多个线程可能在不同的成员函数中被阻塞，但是一次只有一个线程可以进行操作。尽管如此，部分限制来自于在实现中使用 std::queue<>；通过使用标准容器，你基本上有了一个受到保护或者没受到保护的数据项。通过控制数据结构的详细实现，可以提供更细粒度的锁定，并且实现更高级别的并发。

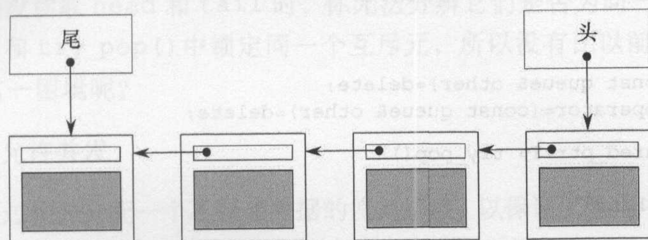



图 6.1　用单链表表示的队列

### 6.2.3　使用细粒度锁和条件变量的线程安全队列

在清单 6.2 和清单 6.3 中，有一个被保护的数据项（data_queue）和一个互斥元。为了使用细粒度锁，你需要瞧一瞧队列的组成部分，并且将一个互斥元与每个不同的数据项联系起来。

实现队列最简单的数据结构为单链表，如图 6.1 所示。队列包含一个头（**head**）指针，指向链表的第一项，并且每一项都指向下一项。将数据项从队列中移走时，首先将头指针指向下一个数据项，然后返回以前头指针所指向的数据项。

数据项被添加到队列的另一端。为了实现这一点，队列还包含一个尾（**tail**）指针，指向链表的最后一项。通过将最后一项的 **next** 指针指向新结点，然后将尾指针更新为指向新的一项，可以实现添加结点。当链表为空时，头指针和尾指针均为 NULL。

清单 6.4 展示了这种队列的简单实现，基于清单 6.2 中队列接口的缩减版本。因为此队列只支持单线程使用，因此只需要一个 try_pop() 函数，而没有 wait_and_pop()。

**清单 6.4　一种简单的单线程队列实现**

```
template<typename T>
class queue
{
private:
    struct node
    {
        T data;
        std::unique_ptr<node> next;

        node(T data_):
```

```
            data(std::move(data_))
        {}
    };

    std::unique_ptr<node> head;   ←❶
    node* tail;                   ←❷

public:
    queue()
    {}

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    std::shared_ptr<T> try_pop()
    {
        if(!head)
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(
            std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head=std::move(head);
        head=std::move(old_head->next);   ←❸
        return res;
    }
    void push(T new_value)
    {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail=p.get();
        if(tail)
        {
            tail->next=std::move(p);   ←❹
        }
        else
        {
            head=std::move(p);   ←❺
        }
        tail=new_tail;   ←❻
    }
};
```

首先，要注意清单6.4使用了std::unique_ptr<node>来管理结点，因为这可以保证当结点不再需要时，它们（以及它们指向的数据）能被删除，而无需显式地编写delete。所有者链表从head开始管理，指向最后一个结点的tail是个裸指针。

尽管在单线程环境中这种实现方式可以良好地工作，但是如果在多线程环境下试图使用细粒度锁，就会带来问题。假定有两个数据项（head❶ 和 tail❷），原则上可以使用两个互斥元，一个用来保护head，另一个用来保护tail，但是这样会存在一些问题。

最明显的问题就是，push()可以修改head❺ 和 tail❻，因此它就需要锁住

这两个互斥元。虽然倒霉，但也不是什么大问题，因为锁住两个互斥元是可能的。关键的问题是，push()和 pop()都要访问某个结点的 next 指针。push()更新 tail->next❹，try_pop()读取 head->next❸。如果队列中只有一个结点，那么 head==tail，即 head->next 和 tail->next 是同一个的对象，因而需要保护。因为还没读取 head 和 tail 时，你无法分辨它们是否为同一个对象，你就必须在 push()和 try_pop()中锁定同一个互斥元，所以没有比以前做得更好。有没有办法摆脱这一困境呢？

## 1. 通过分离数据允许并发

可以通过预先分配一个不存储数据的傀儡结点，以保证了队列中总是至少会有一个结点，将在头尾的两个访问分开，来解决这个问题。对于一个空队列，head 和 tail 都指向这个傀儡结点，而不是 NULL。这样就没问题了，因为如果当队列为空，try_pop()不会访问 head->next。当你将一个结点加入队列（于是就有一个真正的结点），head 和 tail 就指向不同的结点，因此在 head->next 和 tail->next 上就不存在竞争。缺点是，必须添加一个额外的间接层，通过指针存储数据，以便允许假结点。清单 6.5 展示该实现现在的样子。

**清单 6.5 使用傀儡结点的简单队列**

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;        <-❶
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    queue():
        head(new node),tail(head.get())      <-❷
    {}

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    std::shared_ptr<T> try_pop()
    {
        if(head.get()==tail)            <-❸
        {
            return std::shared_ptr<T>();
        }
```

```
        std::shared_ptr<T> const res(head->data);      <-❹
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);                <-❺
        return res;                          <-❻
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));      <-❼
        std::unique_ptr<node> p(new node);              <-❽
        tail->data=new_data;                  <-❾
        node* const new_tail=p.get();
        tail->next=std::move(p);
        tail=new_tail;
    }
};
```

对 try_pop()的改变是很小的。首先，是比较 head 与 tail❸，而不是检查其是否为 NULL，因为傀儡结点意味着 head 不可能是 NULL。因为 head 是一个 std::unique_ptr<node>，你需要调用 head.get()来进行比较。其次，因为 node 现在是通过指针来存储数据的 ❶，所以可以直接获取指针 ❹ 而不必构造 T 的一个新实例。最大改变是在 push()中，必须在堆上创建 T 的一个新实例，并且在 std::shared_ptr<>❼中取得其所有权（使用 std::make_shared 是为了避免第二次为引用计数分配内存的开销）。你创建的新结点将会作为新的傀儡结点，因此无需向构造函数提供 new_value❽。取而代之的是，将旧的傀儡结点上的数据设置为新分配的 new_value 的副本❾。最后，为了得到一个傀儡结点，你必须在构造函数中创建它 ❷。

到目前为止，我敢肯定你想知道这些变化为你带来了什么，并如何让队列变得线程安全。好吧，push()现在只访问 tail，不访问 head，这是一个改进。try_pop()既访问 head 又访问 tail，但只要在最初的比较中需要 tail，因此这个锁是很短暂的。最大的收获就是，傀儡结点意味着 try_pop()和 push()不会在同一个结点上进行操作，因此不再需要一个包含一切的互斥元。因此，你可以为 head 和 tail 各设置一个互斥元。那锁应该放在哪呢？

我们的目标是实现最大程度的并发，因此希望持有锁的时间尽可能的短。push()比较简单：互斥元在访问 tail 的全程都需要被锁定，这意味着应该在新节点被分配好之后❽以及为当前的尾结点分配数据之前❾，锁定互斥元。该锁定需要一直持有到函数结束。

try_pop()就没那么简单了。首先，你需要锁定 head 上的互斥元，并持有它直到 head 使用完毕。本质上，正是这个互斥元决定了哪个线程进行 pop 操作。一旦 head 改变 ❺，你就可以解锁该互斥元；在返回结果的时候 ❻，无需锁定它。剩下对 tail 的访问，需要锁定尾互斥元。因为只需要访问 tail 一次，所以可以只在进行读取的时候获取该互斥元。最好在将其封装在函数内部来实现。实际上，因为需要锁定 head 互斥

元的代码只是该成员的子集，所以将其封装在函数里会更清晰。最终的代码如清单 6.6 所示。

**清单 6.6 使用细粒度锁的线程安全队列**

```
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;
    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return nullptr;
        }
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }
public:
    threadsafe_queue():
        head(new node),tail(head.get())
    {}

    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;

    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_head=pop_head();
        return old_head?old_head->data:std::shared_ptr<T>();
    }

    void push(T new_value)
    {
```

```
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail=p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        tail->next=std::move(p);
        tail=new_tail;
    }
};
```

让我们用批判的眼光来分析这段代码，思考 6.1.1 节中列出的准则。在寻找被破坏的不变量前，你应该确定它们到底是什么。

- tail->next==nullptr。
- tail->data==nullptr。
- head==tail 表明这是一个空链表。
- 只有一个元素的链表必须满足 head->next==tail。
- 对于链表中的每一个结点 x，当 x!=tail 时，x->data 指向 T 的一个实例，并且 x->next 指向链表中的下一个结点。x->next==tail 表明 x 是链表中的最后一个结点。
- 从 head 开始，沿着 next 结点会最终迭代到 tail。

就其本身而言，push()是直观的：对数据结构所做的唯一更改受到 tail_mutex 的保护，并且它们维持了不变量，因为新的尾结点是一个空结点，并且 data 和 next 已经正确设置给旧的尾结点了，而旧的尾结点成为了链表中最后一个真正的结点。

try_pop()这一部分很有趣。结果证明了不仅 tail_mutex 上的锁对于保护 tail 本身的读取是必要的，而且确保从头结点读取数据不会产生数据竞争也是很必要的。如果没有这个互斥元，就有可能出现一个线程调用 try_pop()的同时，另一个线程调用 push()，而且它们的操作并没有确定的顺序。尽管每个成员函数在互斥元上都持有一个锁，但它们锁定的是**不同的**互斥元，并且可能访问相同的数据。毕竟，队列中的所有数据都起源于对 push()的调用。因为线程都可能会访问同一个数据而没有确定的顺序，就有可能导致数据竞争和未定义的行为，正如你在第 5 章中看到的那样。幸亏在 get_tail()中对 tail_mutex 的锁定解决了一切问题。因为调用 get_tail()与调用 push()锁定了相同的互斥元，在这两次调用之间就有了确定的顺序。要么调用 get_tail()发生在调用 push()之前，这种情况下 get_tail()看到的是 tail 的旧值。要么调用 get_tail()发生在调用 push()之后，这种情况下 get_tail()看到的是 tail 的新值，**并且新的数据附在了** tail **之前的值上**。

在锁定 head_mutex 的情况下调用 get_tail()也是很重要的。如果不这么做，对 pop_head()的调用可能会被阻塞在调用 get_tail()和锁定 head_mutex 之间，因为可能有别的线程调用 try_pop()（接下来就是 pop_head()），并且先获得了锁，

从而阻止刚开始的线程继续执行下去。

```
std::unique_ptr<node> pop_head()    <-- 这是一个有缺陷的实现
{
    node* const old_tail=get_tail();    <-- ❶ 在 head_mutex 的锁的外部获取旧的 tail 值
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if(head.get()==old_tail)    <-- ❷
    {
        return nullptr;
    }
    std::unique_ptr<node> old_head=std::move(head);
    head=std::move(old_head->next);    <-- ❸
    return old_head;
}
```

在这种损坏的情况下，对 get_tail() 的调用 ❶ 是在锁的范围之外做出的，你可能会发现在你的初始线程能够获取 head_mutex 上的锁的时候，head 和 tail 都已经发生了变化，并且不仅返回的尾结点不仅仅不再是尾部，甚至都不再是链表的一部分了。这有可能意味着即使 head 真的是最后一个结点，head 和 old_tail 的比较 ❷ 也会失败。因此，当更新 head❸ 的时候，你可能将 head 移动到 tail 之前，超过链表的结尾，破坏数据结构。在清单 6.6 的正确实现中，始终保持在 head_mutex() 上的锁的范围内调用 get_tail()。这就确保没有别的线程可以改变 head，并且 tail 只会移得更远（随着调用 push() 加入新的结点），因此是百分百安全的。head 永远都不会越过 get_tail() 返回的值，因而保持了不变量。

一旦 pop_head() 通过更新 head，将结点从队列中删除，互斥元就解锁了，并且 try_pop() 就可以提取数据并在有结点的时候将其删除（如果没有，就返回一个 std::shared_ptr<>的 NULL 实例），按理说它就是能够访问该结点的唯一线程。

接下来，外部接口是清单 6.2 中的程序的一个子集，因此同样可以分析得到：接口中不存在固有的竞争条件。

异常就更有趣了。因为改变了数据分配模式，因此很多地方都可能产生异常。try_pop() 的操作中只有锁定互斥元才能引发异常，并且直到它获取锁之后才会修改数据。因此 try_pop() 是异常安全的。另一方面，push() 在堆上分配一个 T 的实例以及一个新的结点实例，而这两者都可能会引发异常。不管怎样，这两个新分配的对象都赋值给智能指针，因此当引发异常时它们就会被释放。一旦获取了锁，push() 中的其他操作都不会引发异常，所以你再次稳操胜券，push() 也是异常安全的。

因为没有改变接口，所以没有新的死锁的外部机会。同样也没有内部机会，只有在 pop_head() 中才会获取两个锁，它总是先获取 head_mutex，然后再获取 tail_mutex，所以也不会死锁。

剩下的问题就是关于并发的实际可能性。这种数据结构实际上具有比清单 6.2 中数

据结构相当多的并发范围，因为这些锁是更细粒度的，并且在锁外部完成的更多。例如，在 push()中，分配新结点和新数据项的是无需持有锁的。这就意味着多个线程可以并发地分配新节点和新数据项而不会出现问题。每次只有一个线程可以在链表中插入新结点，并且此操作仅仅是通过一些简单的指针赋值来实现的，所以与所有内存分配操作都在 std::queue<>内部的、基于 std::queue<>的实现相比，它持有锁的时间根本就不算多。

同样，try_pop()持有 tail_mutex 的时间也很短，以保护 tail 的读取。因此，几乎整个对 try_pop()的调用可以与 push()的调用同时发生。同样，当持有 head_mutex 的时候所执行的操作是很少的，昂贵的 delete 操作（在结点指针的析构函数中）在锁的外面。这就增加了同时发生的调用 try_pop()的数量。一次只有一个结点可以调用 pop_head()，但是多个线程可以删除旧结点并且安全地返回数据。

## 2. 等待一个数据项 pop

清单 6.6 提供了一个使用细粒度锁的线程安全队列，但它只支持 try_pop()（并且只有一种重载）。前面清单 6.2 中的有用的 wait_and_pop()函数呢？能否用细粒度锁实现相同的接口呢？

当然，回答是肯定的，但真正的问题是，怎么做？修改 push()是很简单的，只需要在函数的尾部添加 data_cond.notify_one()调用即可，就如同在清单 6.2 中一样。实际上，这并非那么简单，使用细粒度锁是为了实现并发量的最大化。如果在对 notify_one()的调用中保留互斥元被锁定（如同清单 6.2），那么如果被通知的线程在互斥元解锁之前被唤醒，它就得等待互斥元。另一方面，假设在调用 notify_one()之前就解锁互斥元，那么当等待中的线程被唤醒时，此互斥元就可以被它使用（假设没有别的线程先锁定它）。这是一个小小的改进，但某些情况下可能是很重要的。

wait_and_pop()更复杂一些，因为得决定在哪里等待，断言是什么，以及需要锁定哪个互斥元。你所等待的条件是“队列非空”，它是用 head!=tail 表示的。如上所示，这有可能要求 head_mutex 和 tail_mutex 都被锁定，但是在清单 6.6 中，你已经决定只需要在读取 tail 的时候锁住 tail_mutex，比较本身是不需要的，因此可以将相同的逻辑应用到此处。如果将断言设定为 head!=get_tail()，就只需要持有 head_mutex，因此在调用 data_cond.wait()时可以使用 head_mutex 上的锁。一旦增加了等待逻辑，这种实现就跟 try_pop()一样了。

try_pop()的第二个重载以及相对应的 wait_and_pop()重载就需要仔细思考一下。如果只是将从 old_head 获取到的 std::shared_ptr<>替换为向 value 参数拷贝赋值，就可能存在异常安全问题。此刻，数据项已经从队列中删除，且互斥元已解锁，剩下的就是向调用者返回数据。然而，如果拷贝赋值引发异常（这是很有可能发生的），数据项就会丢失，因为无法将其返回到队列中同样的位置。

如果模板参数中使用的实际类型 T 具有不引发异常的移动赋值运算符，或是不引发异常的交换操作，就可以使用之，但是我们实际上更想要一个适用于所有类型 T 的通用解决方案。在这种情况下，就需要在结点从链表中删除前，将可能的异常引发移到锁的范围内。这就意味着，需要另外的 pop_head() 重载，在修改链表之前获取存储的值。

相比之下，empty() 就很平常了，只需要锁定 head_mutex 并检查 head==get_tail()（如清单 6.10 所示）。队列最终的代码如清单 6.7、清单 6.8、清单 6.9 和清单 6.10 所示。

**清单 6.7　使用锁和等待的线程安全队列：内部与接口**

```
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;
    std::condition_variable data_cond;
public:
    threadsafe_queue():
        head(new node),tail(head.get())
    {}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;

    std::shared_ptr<T> try_pop();
    bool try_pop(T& value);
    std::shared_ptr<T> wait_and_pop();
    void wait_and_pop(T& value);
    void push(T new_value);
    void empty();
};
```

向队列中入队新结点是很直观的——其实现（如清单 6.8 所示）与之前展示的非常接近。

**清单 6.8　使用锁和等待的线程安全队列：push 新值**

```
template<typename T>
void threadsafe_queue<T>::push(T new_value)
{
    std::shared_ptr<T> new_data(
        std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
```

```
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        node* const new_tail=p.get();
        tail->next=std::move(p);
        tail=new_tail;
    }
    data_cond.notify_one();
}
```

就像一直提到的那样，复杂性都在 **pop** 端，要利用一系列的辅助函数来简化问题。清单 6.9 展示了 wait_and_pop()是如何实现的以及相关的辅助函数。

**清单 6.9　使用锁和等待的线程安全队列：wait_and_pop()**

```
template<typename T>
class threadsafe_queue
{
private:
    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head()    <-- ❶
    {
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }

    std::unique_lock<std::mutex> wait_for_data()    <-- ❷
    {
        std::unique_lock<std::mutex> head_lock(head_mutex);
        data_cond.wait(head_lock,[&]{return head.get()!=get_tail();});
        return std::move(head_lock);    <-- ❸
    }

    std::unique_ptr<node> wait_pop_head()
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());    <-- ❹
        return pop_head();
    }

    std::unique_ptr<node> wait_pop_head(T& value)
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());    <-- ❺
        value=std::move(*head->data);
        return pop_head();
    }
public:
    std::shared_ptr<T> wait_and_pop()
    {
```

```
        std::unique_ptr<node> const old_head=wait_pop_head();
        return old_head->data;
    }

    void wait_and_pop(T& value)
    {
        std::unique_ptr<node> const old_head=wait_pop_head(value);
    }
};
```

清单 6.9 所示的 pop 端实现具有一些辅助函数来简化代码和去重，例如 pop_head() ❶ 与 wait_for_data() ❷。相对应地，它们修改链表以删除首项，并且等待队列中有数据要 pop。wait_for_data() 特别值得一提，因为它不仅使用一个 lambda 函数作为断言来等待条件变量，而且它向调用者返回锁的实例 ❸。这是为了确保当数据被相关的 wait_pop_head() 重载 ❹、❺ 修改时，持有相同的锁。在清单 6.10 中列出的 try_pop() 代码中也复用了 pop_head()。

**清单 6.10 使用锁和等待的线程安全队列：try_pop()和 empty()**

```
template<typename T>
class threadsafe_queue
{
private:
    std::unique_ptr<node> try_pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        return pop_head();
    }

    std::unique_ptr<node> try_pop_head(T& value)
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        value=std::move(*head->data);
        return pop_head();
    }
public:
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_head=try_pop_head();
        return old_head?old_head->data:std::shared_ptr<T>();
    }

    bool try_pop(T& value)
```

```
    {
        std::unique_ptr<node> const old_head=try_pop_head(value);
        return old_head;
    }
    void empty()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        return (head.get()==get_tail());
    }
};
```

这种队列的实现将会作为第7章中提到的无锁队列的基础。这是一个无界队列。只要还有可用的内存，即使没有值被删除，线程也可以一直往队列中 push 新的值。与无界队列相对的是**有界**队列，当队列被创建的时候，它的最大长度也已经定下来了。一旦一个有界队列已满，再试图往队列中 push 更多的元素就会失败或者阻塞，直到有元素从队列中 pop 出，以腾出空间。有界队列对那些基于待执行的任务在线程间划分工作，要确保均匀铺开工作的时候，是很有用的（参见第 8 章）。这可以阻止线程过快填充队列，以至于远远超过从队列中读取数据的线程。

这里展示的无界队列的实现可以通过在push()中等待条件变量，很容易地扩展为限制长度的队列。不同于等待队列中有数据项（例如在pop()中所做的），你需要等待队列中的项目数量低于最大值。对有界队列的进一步讨论超出了本书的范围。现在，让我们越过队列，来看一看更复杂的数据结构。

## 6.3 设计更复杂的基于锁的数据结构

栈和队列是很简单的，它们的接口极其有限，并且紧紧关注特定的目的。并非所有的数据结构都是那么简单的，大部分数据结构支持各种操作。原则上，这可能导致更多的并发机会，但因为需要考虑多种访问模式，使得保护数据的任务变得更加困难。当为并发访问设计数据结构时，能够执行的各种操作的精确特性是很重要的。

为了研究所涉及到的问题，我们先来看看查找表的设计。

### 6.3.1 编写一个使用锁的线程安全查找表

查找表或字典将一种类型（键类型）的值与另外一种相同或不同类型（映射类型）的值联系起来。一般来说，这种数据结构的目的是使代码可以用一个给定的键值来查询相关的数据。在 C++标准库中，是通过使用关联容器器来实现这种功能的，例如，std::map<>、std::multimap<>、std::unordered_map<>以及std::unordered_multimap<>。

查找表的使用模式与栈和队列都不同。栈和队列上的每个操作都会在一定程度上

对它有所修改，要么添加一个元素要么删除一个元素，而查找表则很少会被修改。清单 3.13 中的简单 DNS 缓存就是这种情形的一个例子，与 std::map<>相比，它的接口极大地简化了。如你在栈和队列中看到的，当从多个线程并发访问数据结构时，标准容器的接口并不合适，因为在接口设计中存在固有的竞争条件，所以它们需要被削减并修订。

从并发的角度来说，std::map<>接口的最大问题就是迭代器。尽管当别的线程访问（以及修改）容器时，拥有一个能够安全访问容器的迭代器也是有可能的，但这很棘手。正确把握迭代器要求你去处理以下的问题，例如另一个线程正在删除迭代器引用的元素，这很麻烦。作为线程安全查找表要砍掉的第一个接口，你应跳过迭代器。std::map<>（以及标准库中其他的关联容器）的接口在很大程度上基于迭代器，所以将它们踢到一边并且重新开始设计接口可能是值得的。

查找表只有一些的基本操作。

- 添加新的键/值对。
- 改变与给定的键相关联的值。
- 删除键及其关联的值。
- 获得与给定键相关联的值，如果有的话。

还有一些容器范围的操作也是有用的，例如检查容器是否为空，键的完整列表的快照，或是键/值对的完整集合的快照。

如果坚持简单的线程安全准则，例如不返回引用，以及在每个成员函数上都有一个互斥元，那么这些操作都是安全的。它们要么出现在其他线程的某个修改之前，要么在之后。最有可能产生竞争条件的，是在添加一个新的键/值对的时候。如果两个线程添加一个新值，只有一个线程会胜出，第二个会因此而失败。一种可能的方法将添加和改变操作整合进单个成员函数中，就像你为清单 3.13 中的 DNS 缓存所做的那样。

从接口的角度来看，有趣一点是获取相关联值时的"如果有"的部分。一种选择是当键不存在的情况下，允许用户提供一个"默认的"结果来返回。

```
mapped_type get_value(key_type const& key, mapped_type default_value);
```

在这种情况下，如果没有显式提供 default_value，可以使用 mapped_type 的默认构造函数实例。这也可以扩展为返回一个 std::pair<mapped_type,bool>类型的实例，而不只是 mapped_type 的实例，这里的 bool 指示值是否存在。另一种选择就是，返回一个引用该值的智能指针。如果指针的值为 NULL，就是没有返回值。

如上所述，一旦决定了接口，那么（假设没有接口竞争条件）可以通过在每个成员函数中使用一个互斥元和一个简单锁来保护下层的数据结构，以保证线程安全。然而，

这会浪费通过独立的函数来读取数据结构并修改它所提供的并发可能性。一种方法是使用一个支持多个读线程或者一个写线程的互斥元，例如清单 3.13 中使用的 boost::shared_mutex。尽管这种方法可以提高并发访问的可能性，但是每次只有一个线程能够修改数据结构。理想情况下，你会想要做得更好一些。

### 设计一个细粒度锁的 MAP 数据结构

如同在 6.2.3 中提到的队列一样，为了允许细粒度锁，你需要仔细考虑数据结构的细节，而不是仅仅封装一个类似于 std::map<>这样已存在容器。这里通常有三种常见的方法来实现一个类似于查找表的关联容器。

- 二叉树，例如红黑树。
- 已排序数组。
- 哈希表。

二叉树不能为扩大并发机会提供太多的空间，每次查找或修改必须从访问根节点开始，因此根节点必须被锁定。尽管当线程沿着树往下移动的时候会释放这个锁，但是这也不比锁定整个数据结构好多少。

已排序数组就更糟了，因为无法事先得知一个给定的数据在数组的哪个位置，所以就需要一次锁定整个数组。

只剩下哈希表了。假设有一定数量的桶，一个键属于哪个桶完全取决于这个键及其哈希函数的特性。这就意味着可以安全地在每个桶上有一个独立的锁。如果再次使用支持多重读或单一写的互斥元，你就将并发机会增加了 N 重，这里的 N 是桶的数量。其缺点是为键找一个好的哈希函数。C++标准库提供了 std::hash<>模板，可以用来实现这一目的。它已经为 int 等基本类型以及 std::string 这样的常见库类型进行了特化，而且使用者也可以轻易地为别的键类型将其进行特化。如果效仿标准的无序容器，以及在进行哈希计算时将函数对象类型作为模板参数，那么使用者可以选择是否为键类型特化 std::hash<>，或是提供一个单独的哈希函数。

那么，让我们来看清单 6.11 中的代码。一个线程安全查找表的实现将会是怎样的呢？这里列出了一种可能的实现。

**清单 6.11 线程安全查找表**

```
template<typename Key,typename Value,typename Hash=std::hash<Key> >
class threadsafe_lookup_table
{
private:
    class bucket_type
    {
    private:
        typedef std::pair<Key,Value> bucket_value;
```

```
    typedef std::list<bucket_value> bucket_data;
    typedef typename bucket_data::iterator bucket_iterator;

    bucket_data data;
    mutable boost::shared_mutex mutex;  <-❶

    bucket_iterator find_entry_for(Key const& key) const  <-❷
    {
        return std::find_if(data.begin(),data.end(),
            [&](bucket_value const& item)
            {return item.first==key;});
    }
public:
    Value value_for(Key const& key,Value const& default_value) const
    {
        boost::shared_lock<boost::shared_mutex> lock(mutex);  <-❸
        bucket_iterator const found_entry=find_entry_for(key);
        return (found_entry==data.end())?
            default_value:found_entry->second;
    }

    void add_or_update_mapping(Key const& key,Value const& value)
    {
        std::unique_lock<boost::shared_mutex> lock(mutex);  <-❹
        bucket_iterator const found_entry=find_entry_for(key);
        if(found_entry==data.end())
        {
            data.push_back(bucket_value(key,value));
        }
        else
        {
            found_entry->second=value;
        }
    }

    void remove_mapping(Key const& key)
    {
        std::unique_lock<boost::shared_mutex> lock(mutex);  <-❺
        bucket_iterator const found_entry=find_entry_for(key);
        if(found_entry!=data.end())
        {
            data.erase(found_entry);
        }
    }
};

std::vector<std::unique_ptr<bucket_type> > buckets;  <-❻
Hash hasher;

bucket_type& get_bucket(Key const& key) const  <-❼
{
    std::size_t const bucket_index=hasher(key)%buckets.size();
    return *buckets[bucket_index];
}

public:
    typedef Key key_type;
```

```
    typedef Value mapped_type;
    typedef Hash hash_type;

    threadsafe_lookup_table(
        unsigned num_buckets=19,Hash const& hasher_=Hash()):
        buckets(num_buckets),hasher(hasher_)
    {
        for(unsigned i=0;i<num_buckets;++i)
        {
            buckets[i].reset(new bucket_type);
        }
    }

    threadsafe_lookup_table(threadsafe_lookup_table const& other)=delete;
    threadsafe_lookup_table& operator=(
        threadsafe_lookup_table const& other)=delete;
    Value value_for(Key const& key,
                    Value const& default_value=Value()) const
    {
        return get_bucket(key).value_for(key,default_value);    <-8
    }
    void add_or_update_mapping(Key const& key,Value const& value)
    {
        get_bucket(key).add_or_update_mapping(key,value);    <-9
    }
    void remove_mapping(Key const& key)
    {
        get_bucket(key).remove_mapping(key);    <-10
    }
};
```

该实现使用 std::vector<std::unique_ptr<bucket_type>>❻ 来持有桶，允许在构造函数中指定桶的数量。默认值是 19，这是一个任意选择的质数。哈希表与质数数量的桶合作得最好。每个桶都被一个 boost::shared_mutex 的实例保护❶，使得每个桶都可以允许很多个并发读或者单个调用其中一个修改函数。

因为桶的数量是固定的，所以在调用 get_bucket()函数❼时无需锁定❽、❾、❿，而且桶互斥元随后可以被锁定为共享（只读）所有权❸，或是独占（读/写）所有权❹、❺，对每个函数都是适用的。

这三个函数都使用桶上的 find_entry_for()成员函数❷来确定在桶上是否有入口。每个桶包含一个键/值对的 std::list<>，因此添加和删除项目是很简单的。

我还关注了并发的角度，所有的东西都被互斥元锁合适地保护着，那么异常安全呢？value_for 并不修改任何东西，所以没关系，如果它引发异常，也不会影响数据结构。remove_mapping 调用 erase 的时候会修改列表，但是保证不会引发异常，因此是安全的。只有 add_or_update_mapping 在 if 的两个分支上可能会引发异常。push_back 是异常安全的，当它引发异常时，会将列表留在原始状态，因此这个分支

是没问题的。在这种情况下，唯一的问题就是当替换一个现存值时所做的赋值，如果此赋值引发异常，那么就得指望它不要修改原始值。然而，这在整体上并不影响数据结构，而且完全是用户提供类型的属性，因此可以安全地把它留给用户来处理。

在本节的开头，我提到这种查找表的一个不错的功能，就是可以获取到当前状态的快照的选项，比如 std::map<>。为了确保能够得到该状态的一致的副本，就需要锁定整个容器，也就是需要锁定所有的桶。因为查找表中"普通的"操作一次只需要锁定一个桶，这就会是唯一一个需要锁定所有桶的操作。因此，只要每次按照相同的顺序（例如，递增桶的索引）来锁定桶，就不会有死锁的机会。清单 6.12 给出了一个实现。

**清单 6.12 获取 threadsafe_lookup_table 的内容作为一个 std::map<>**

```
std::map<Key,Value> threadsafe_lookup_table::get_map() const
{
    std::vector<std::unique_lock<boost::shared_mutex> > locks;
    for(unsigned i=0;i<buckets.size();++i)
    {
        locks.push_back(
            std::unique_lock<boost::shared_mutex>(buckets[i].mutex));
    }
    std::map<Key,Value> res;
    for(unsigned i=0;i<buckets.size();++i)
    {
        for(bucket_iterator it=buckets[i].data.begin();
            it!=buckets[i].data.end();
            ++it)
        {
            res.insert(*it);
        }
    }
    return res;
}
```

清单 6.11 中查找表实现通过单独锁定每个桶以及使用一个 boost::shared mutex 实例来允许基于每个桶的并发读取，这就从总体上增加了查找表的并发机会。但是如果要通过更细粒度的锁来增加并发的潜力呢？在下一节，将通过使用具有迭代器支持的线程安全链表容器来实现。

## 6.3.2 编写一个使用锁的线程安全链表

链表是一种最基本的数据结构，因此它应该能被直接写成线程安全的，不是吗？那么，这取决于你追求什么样的功能，并且需要提供迭代器支持，这是我一直避免将其加入到你的基础图中的东西，因为它太复杂了。STL 风格的迭代器支持，指的是迭代器必须持有某种对容器内部数据结构的引用。如果容器可以被另一个线程修改，这个引用必须仍然有效，这就从根本上要求迭代器在部分数据结构上持有锁。考虑到 STL 风格的迭代

代器的生存期是完全不受容器控制的，这就不是个好主意。

另一种方式是提供类似于 for_each 这样的迭代函数作为容器本身的一部分。这就让容器完全负责迭代器和锁，但是这与第 3 章中提到的避免死锁原则是冲突的。为了使得 for_each 做一些有用的操作，它就必须在持有内部锁的时候调用用户提供的代码。不仅如此，为了使用户提供的代码能够作用于数据项，它必须将对每个数据项的引用传递给用户提供的代码。你可以通过向用户提供的代码传递每个数据项的副本来避免这一点，但是当数据项很大时，这种方式就很耗费资源。

因此，目前我们把它留给用户，让他们确保不会在用户提供的操作中因获取锁而导致死锁，并且通过在锁外的访问中存储引用以避免导致数据竞争。就查找表所使用的链表来说，它是完全安全的，因为不会做任何不恰当的操作。

留给你的问题是，要为链表提供哪些操作。如果回顾一下清单 6.11 以及 6.12，就可以知道需要下列操作。

- 向链表添加新项目。
- 从链表中删除满足一定条件的项目。
- 在链表中查找满足一定条件的项目。
- 更新满足一定条件的项目。
- 复制链表中每个项目到另一个容器中。

为了令其成为良好的通用链表容器，添加进一步的操作例如在指定位置插入是很有用的，但是对于查找表是不需要的，因此我将它留给读者作为练习。

在链表中使用细粒度锁的基本思想是每个结点使用一个互斥元。如果链表很大，就会有很多互斥元！其好处就是在链表不同部分的操作是真正并发的。每个操作仅在其真正关注的结点上持有锁，并且当它移动到下一个结点时，会解锁每个结点。清单 6.13 给出了正是这样一个链表的实现。

**清单 6.13　支持迭代的线程安全链表**

```
template<typename T>
class threadsafe_list
{
    struct node                    <-❶
    {
        std::mutex m;
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
        node():                    <-❷
            next()
        {}
                                   ❸
        node(T const& value):      <-┘
            data(std::make_shared<T>(value))
        {}
```

```
    };

    node head;

public:
    threadsafe_list()
    {}

    ~threadsafe_list()
    {
        remove_if([](node const&){return true;});
    }

    threadsafe_list(threadsafe_list const& other)=delete;
    threadsafe_list& operator=(threadsafe_list const& other)=delete;

    void push_front(T const& value)
    {
        std::unique_ptr<node> new_node(new node(value));      ⟵❹
        std::lock_guard<std::mutex> lk(head.m);
        new_node->next=std::move(head.next);                ⟵❺
        head.next=std::move(new_node);       ⟵❻
    }

    template<typename Function>
    void for_each(Function f)         ⟵❼
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m)          ⟵❽
        while(node* const next=current->next.get())        ⟵❾
        {
            std::unique_lock<std::mutex> next_lk(next->m);    ⟵❿
            lk.unlock();         ⟵⓫
            f(*next->data);    ⟵⓬
            current=next;
            lk=std::move(next_lk);    ⟵⓭
        }
    }

    template<typename Predicate>
    std::shared_ptr<T> find_first_if(Predicate p)        ⟵⓮
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next.get())
        {
            std::unique_lock<std::mutex> next_lk(next->m);
            lk.unlock();
            if(p(*next->data))    ⟵⓯
            {
                return next->data;    ⟵⓰
            }
            current=next;
            lk=std::move(next_lk);
        }
        return std::shared_ptr<T>();
    }
```

```
    template<typename Predicate>
    void remove_if(Predicate p)                    ⑰
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next.get())
        {
            std::unique_lock<std::mutex> next_lk(next->m);
            if(p(*next->data))                     ⑱
            {
                std::unique_ptr<node> old_next=std::move(current->next);
                current->next=std::move(next->next);          ⑲
                next_lk.unlock();
            }                                       ⑳
            else
            {
                lk.unlock();                        ㉑
                current=next;
                lk=std::move(next_lk);
            }
        }
    }
};
```

清单 6.13 中的 threadsafe_list<>是一个单链表，每个入口是一个 node 结构体 ❶。默认构造的 node 是链表的 head，开始时它的 next 指针为 NULL❷。新结点是通过 push_front()函数增加的，首先构造一个新结点 ❹，在堆上分配存储的数据 ❸，保留 next 指针为 NULL。然后你需要为 head 结点获取互斥元上的锁来得到正确的 next 值 ❺，并且通过将 head.next 设置为指向新结点来实现将这个结点插入链表前方 ❻。到目前为止，一切都还不错，你只需要锁住一个互斥元来将新项目插入链表，因此没有死锁的风险。同样，缓慢的内存分配发生在锁之外，因此锁只保护几个指针值的更新并且不会失败。下面是迭代函数。

首先，我们来看看 for_each()❼。这个操作用接受某种类型的 Function，作用于表中的每一个元素；通常在大多数标准库中，它会通过值形式接受此函数，并且可以与真正的函数或者是具有函数调用操作符的某种类型的对象一起运作。在这种情况下，函数必须接受类型为 T 的值作为唯一的参数。这就是你进行交替锁定的地方。刚开始，你锁定 head 结点上的互斥元❽。然后就可以安全地获得指向 next 结点的指针（使用 get()因为你并不打算获取该指针的所有权）。如果该指针不为 NULL❾，就锁定那个结点上的互斥元❿来处理数据。一旦你锁定那个结点，就可以释放之前结点的锁⓫，并且调用指定的函数⓬。一旦函数完成，就可以更新 current 指针指向你刚刚处理的结点，并且将锁的拥有权从 next_lk 移动（move）到 lk⓭。因为 for_each 将每个数据项直接传递给所提供的 Function，如果需要的话，你可以使用它更新数据项或者将它们复制到另一个容器中，或其他任何事情。如果函数表现良好这就是安全的，因为拥有数

据项的结点在整个调用中都持有互斥元。

find_first_if()⓮与 for_each()是类似的，最主要的不同之处在于提供的 Predicate 必须返回 true 来表明找到了匹配项或者 false 来表明没找到匹配项⓯。一旦你找到匹配项，你就返回找到的数据⓰而不是继续查找。你可以用 for_each()来实现它，但是一旦找到匹配项就不需要继续处理链表中剩下的部分了。

remove_if()⓱有些不同，因为这个函数必须真正地更新链表，你不能用 for_each()来实现它。如果 Predicate 返回 true⓲，你就通过更新 current->next⓳将这个结点从链表中删除。一旦完成，就可以释放 next 结点互斥元所持有的锁。当你将它移入的 std::unique_ptr<node>超出范围时，该结点就被删除了⓴。在这种情况下，你不用更新 current，因为需要检查新的 next 结点。如果 Predicate 返回 false，你就像以前一样处理㉑。

那么，这些互斥元上存在着死锁或者竞争条件么？答案毫无疑问是否定的，前提是所提供的断言和函数是表现良好的。迭代总是按照同一方式，通常从 head 结点开始，并且总是在释放当前互斥元前就锁住下一个互斥元，因此不可能在不同的线程间有不同的锁顺序。唯一可能产生竞争条件的，就是在 remove_if()中对待删除结点的删除 ，因为你会在解锁互斥元之后才这么做（销毁一个锁定的互斥元是未定义的行为）。然而，稍加思考就会知道这确实是安全的，因为你仍然持有之前结点（current）上的互斥元，因此没有新线程可以试图获取你要删除的结点上的锁。

并发的机会又如何呢？这种细粒度锁的关键是在单个互斥元上提高并发的可能性，那么实现这一点了么？是的，已经实现了。不同的线程可以同时在链表的不同结点上工作，无论它们正在用 for_each()处理每个数据项，还用 find_first_if()搜索，还是用 remove_if()来删除项目⓴。但是因为每个结点的互斥元轮流被锁定，线程不能互相超越。如果一个线程花了很长时间处理一个特定的结点，当别的线程到达此特定结点时就必须等待。

## 6.4 小结

本章开头考虑了为并发设计一个数据结构意味着什么，并且提供了一些准则来实现。然后我们完成一些通用的数据结构（栈、队列、哈希映射以及链表），考虑了如何在设计并发存取的时候应用这些准则来实现它们，使用锁来保护数据并阻止数据竞争。现在你可以考虑设计你自己的数据结构，观察哪里有并发的机会以及哪里可能会存在竞争条件。

在第 7 章，我们将看一看完全避免锁的方法，使用底层原子操作来提供必要的顺序限制，并且遵循同样的准则。

# 第 7 章 设计无锁的并发数据结构

本章主要内容

- 为无需使用锁的并发而设计的数据结构的实现
- 在无锁数据结构中管理内存的技术
- 有助于编写无锁数据结构的简单准则

上一章中，我们分析了为实现并发性设计数据结构时需要考虑的一般方面，考虑了这种设计确保安全的准则。然后，我们验证了几种常见的数据结构，并且分析了使用互斥元和锁来保护共享数据的实现的例子。在前面的几个例子中，使用一个互斥元来保护整个数据结构。在后面的几个例子中，使用多个互斥元来保护数据结构的多个小部分，并且在访问数据结构时允许了更大级别的并发。

互斥元是保证多个线程可以安全访问数据结构，而不会遇到竞争条件和破坏不变量的有效机制。在探讨使用它们的代码的行为时也相对较简单，代码要么让保护数据的互斥元锁定，要么就不这样。然而，这也并不全然那么好。第 3 章中，看到了锁的不当使用会如何导致死锁，并且在基于锁的队列和查找表的例子中，可以看出锁的粒度是如何影响真正并发的潜能。如果能设计出不使用锁就能实现安全并发存取的数据机构，就有可能避免这些问题。这种数据结构被称为**无锁数据结构**。

在本章中，我们将考虑如何将第 5 章中提到的原子操作的内存顺序特性应用到无锁数据结构的构造中。设计这种数据结构时需要特别小心，因为很难做得正确，并且导致设计失败的条件可能很少发生。首先，我们来了解数据结构无锁的含义。然后，在分析

一些实例之前，我们会介绍使用它们的原因，并得出一些通用准则。

# 7.1　定义和结果

使用互斥元、条件变量以及 future 来同步数据的算法和数据结构被称为**阻塞**（**blocking**）的算法和数据结构。调用库函数的应用会中断一个线程的执行，直到另一个线程执行一个动作。这种库函数调用被称为**阻塞**调用，因为直到阻塞被释放时线程才能继续执行下去。通常，操作系统会完全阻塞一个线程（并且将这个线程的时间片分配给另一个线程）直到另一个线程执行了适当的动作将其**解锁**，可以是解锁互斥元、通知条件变量或者使得 future 就绪。

不使用阻塞库函数的数据结构和算法被称为**非阻塞**（**nonblocking**）的。但是，并不是所有这样的数据结构都是无锁（**lock-free**）的，因此我们来看一看非阻塞数据结构的各种类型。

## 7.1.1　非阻塞数据结构的类型

第 5 章中，我们实现了一种使用 `std::atomic_flag` 作为自旋锁的基本互斥元。清单 7.1 中复刻了该代码。

**清单 7.1　使用 std::atomic_flag 的自旋锁互斥元的实现**

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex():
        flag(ATOMIC_FLAG_INIT)
    {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock()
    {
        flag.clear(std::memory_order_release);
    }
};
```

这段代码不调用任何阻塞函数，`lock()`一直循环直到对 `test_and_set()`的调用返回 `false`。这就是**自旋锁**（**spin lock**）名称的由来——代码围绕着循环“旋转”。无论如何，这里没有阻塞调用，因此任何使用此互斥元来保护共享数据的代码因而都是非阻塞的。然而，它并非无锁的。它仍然是一个互斥元，并且一次仍然

只能被一个线程锁定。我们来看看无锁的定义，这样就能明白哪些类型的数据结构是被涉及的。

### 7.1.2　无锁数据结构

对于有资格称为无锁的数据结构，就必须能够让多于一个线程可以并发地访问此数据结构。这些线程不需要做相同的操作，无锁队列可以允许一个线程 push 的同时另一个线程 pop，但是如果两个线程同时试图插 push 新数据项的时候，就会打破无锁队列。不仅如此，如果一个访问数据结构的线程在操作中途被调度器挂起的话，别的线程必须仍然能够完成操作而无需等待挂起的线程。

在数据结构上使用比较/交换操作的算法经常在其中包含循环。使用比较/交换操作是因为有可能另一个线程正在同时修改数据，这种情况下，代码就需要在试图重新比较/交换前重做部分操作。如果比较/交换操作最终在其他线程都被中断的情况下成功，那么这种代码仍然是无锁的。如果没有，最起码要使用自旋锁，是非阻塞的而不是无锁的。

具有这种循环的无锁算法可能会导致一个线程承受饥饿。如果另一个线程在“错误的”时间执行操作，那么当第一个线程持续重试其操作时，别的线程则可以继续前进。能够避免此类问题的数据结构是无等待，也是无锁的。

### 7.1.3　无等待的数据结构

无等待的数据结构是一种无锁的数据结构，并且有着额外的特性，每个访问数据结构的线程都可以在有限数量的步骤内完成它的操作，而不用管别的线程的行为。因为其他线程的冲突而可能卷入无限次重试的算法不是无等待的。

正确地编写无等待的数据结构是极其困难的。为了确保每个线程都能够在有限步骤内完成它的操作，就必须保证每个操作都可以在一个操作周期内执行，并且一个线程执行的操作不会导致另一个线程上操作的失败。这会使得各种操作的整体算法变得相当复杂。

鉴于正确地设计无锁或无等待数据结构是如此困难，你需要一些很好的理由来支撑这一点；你需要确信收益胜于代价。所以，让我们来检验影响此平衡的重点。

### 7.1.4　无锁数据结构的优点与缺点

到了这一步，使用无锁数据结构的最主要的原因就是为了实现最大程度的并发。对于基于锁的容器，总是有可能一个线程必须阻塞，并在可以继续前等待另一个线程完成其操作。互斥元锁的目的就是通过互斥来阻止并发。使用无锁数据结构时，某些线程一

步步地执行操作。使用无等待数据结构时，不管别的线程在做什么操作，每个线程都可以继续执行而不需要等待。这是一种很希望得到但是却很难得到的特性。都很容易在编写基本的一个自旋锁时告终。

使用无锁数据结构的第二个原因是健壮性。当一个线程在持有锁的时候终止，那个数据结构就永远被破坏了。但是如果一个线程在操作无锁数据结构时终止了，就不会丢失任何数据，除了此线程的数据之外，其他线程可以继续正常执行。

另一方面，如果不能排除线程访问数据结构，那么就必须确保持有不变量或选择可以持有的替代的不变量。并且，必须注意你加于操作上的顺序限制。为了避免与数据竞争有关的未定义行为，你就必须在修改时使用原子操作。仅仅如此还是不够的，你必须确保这个改变以正确的顺序对其他线程是可见的。所有这些都意味着设计线程安全数据结构时，不使用锁比使用锁要困难的多。

因为不使用任何锁，因此无锁数据结构是不会发生死锁的，尽管有可能存在活锁。当两个线程都试图修改数据结构，但是对于每个线程来说，另一个线程所做的修改都会要求此线程的操作重新被执行，因此这两个线程都会一直循环和不断尝试，在这种情况下就会发生活锁。除非某个线程先到达（通过协议，通过更快，或完全靠运气），不然此循环会一直继续下去。在这个简单的例子中，活锁通常是短暂的，因为它们取决于线程的精确调度。因此，活锁会降低性能而不会导致长期的问题，但是也是需要注意的事情。根据定义，无等待的代码无法忍受活锁，因为它执行操作的步骤数通常是有上限的。另一方面，这种算法比别的算法更复杂，并且即使当没有线程存取数据结构的时候也需要执行更多的步骤。

这就带来了无锁和无等待代码的另一个缺点，尽管它可以增加在数据结构上操作的并发能力，并且减少了线程等待的时间，但是它可能降低整体的性能。首先，无锁代码使用的原子操作可能比非原子操作要慢很多。并且与基于锁数据结构的互斥元锁代码相比，无锁数据结构中需要更多的原子操作。不仅如此，硬件必须在存取同样的原子变量的线程间同步数据。正如你将在第 8 章中看到的，与多个线程存取同样的原子变量相关的乒乓缓存可能会成为一种显著的性能消耗。总而言之，在选择任何一种方式前，检查基于锁的数据结构和无锁数据结构的相关性能方面（是否为最坏等待时间，平均等待时间，总的执行时间，或其他方面）是很重要的。

下面我们来看一些例子。

## 7.2 无锁数据结构的例子

为了展示在设计无锁数据结构时使用的一些技术，我们来看一些简单数据结构的无锁实现。我们不仅举例子描述了一系列有用的数据结构的实现，而且将举例子强调无锁数据结构设计中比较特殊的部分。

就像之前提到的，依赖于使用原子操作的无锁数据结构，以及与之相关联的内存顺序保证是为了确保数据以正确的顺序对其他线程可见。起初，我们为所有原子操作都使用默认的 memory_order_seq_cst 内存顺序，因为这是最简单的（记住所有的 memory_order_seq_cst 操作构成了全局顺序）。但是在后来的例子中，我们考虑降低一些排序约束到 memory_order_acquire、memory_order_release，甚至 memory_order_relaxed 中。尽管在这些例子中都未直接使用互斥元锁，但是需要记住的是，只有 std::atomic_flag 保证在实现中是不使用锁的。在一些平台上，有些看上去无锁的代码，实际上却可能使用了 C++标准库实现的内部锁（参见第 5 章）。在这些平台上，一个简单的基于锁的数据结构可能更适合。但是还有比这更重要的是，在选择一种实现的时候，必须先确定你的要求，然后考虑有哪些选择可以满足此要求。

因此，我们追溯到一种最简单的数据结构：栈。

### 7.2.1 编写不用锁的线程安全栈

栈的基本假设是相当简单的，按照添加结点的逆序来检索结点——后进先出(LIFO)。因此，确保一次只添加一个值到栈中是很重要的。另一个线程可以立刻检索结点，并且确保只有一个线程返回给定的数值是很重要的。最简单的栈是一个链表，head 指针指向第一个结点（这个结点将被第一个检索），并且每个结点都按顺序指向下一个结点。

在这种原则下，添加一个节点相对比较简单。

1 创建一个新结点。
2 将它的 next 指针指向当前的 head 结点。
3 将 head 结点指向此新结点。

在单线程环境中，这种方法是可以的。但是如果别的线程也在修改栈，那么这种方法就不行了。最重要的是，如果两个线程同时添加结点，那么在第 2 步和第 3 步间就会存在竞争条件。当你的线程在第 2 步读取头结点和第 3 步更新头结点之间，第二个线程可能会修改 head 的值。这就会导致另一个线程所做的修改无效或者有更坏的影响。在我们考虑解决这一竞争条件之前，请注意一旦 head 被更新指向你新创建的结点，别的线程就可以读取该结点。因此，在 head 指向新结点之前将新结点完全准备好也是至关重要的，以后就无法修改此结点了。

那么，我们能够如何处理此竞争条件呢？答案就是在第 3 步中使用一个原子比较/交换操作来保证从你在第 2 步中读取它开始，head 就未被修改过。如果 head 被修改了，那么可以循环和再次尝试。清单 7.2 给出了如何实现不使用锁的线程安全 push()。

**清单 7.2 实现不使用用锁的线程安全 push()**

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        T data;
        node* next;
        node(T const& data_):    ←❶
          data(data_)
        {}
    };

    std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data);    ←❷
        new_node->next=head.load();                    ❸
        while(!head.compare_exchange_weak(new_node->next,new_node));    ←❹
    }
};
```

这段代码恰好符合了上面提到的三点计划：创建一个新节点❷，将新结点的 next 指针指向当前的 head❸，将 head 指向这个新结点❹。通过在 node 构造函数❶中填充 node 结构体的数据，这就可以保证 node 一被构造好就可以被使用，因此这个简单的问题就被解决了。然后使用 compare_exchange_weak()来确保 head 指针的值与 new_node->next❸的值是一样的。如果这两个值是一样的，那么将 head 指向 new_node。这段代码中使用了比较/交换函数的一部分，如果它返回 false 则表明此次比较没有成功（例如，因为另一个线程修改了 head）。此时，第一个参数(new_node->next) 的值将被更新为 head 当前的值。因此，通过此次循环，就不需要你每次重载 head，因为编译器已经做了此项操作。同样，因为失败时只需要直接循环，因此可以使用 compare_exchange_weak 。在某些架构下，它比 compare_exchange_strong 能够产生更优化的代码（如第 5 章所示）。

因此，在没有 pop()操作的情况下，先按照准则来快速检查一下 push()。唯一能引发异常的地方就是构造新结点的时候❶。但是它之后会清除这些，并且链表没有被修改，因此这是非常安全的。因为你建造的数据将被存储为 node 的一部分，并且可以使用 compare_exchange_weak()来更新 head 指针，因此这里没有成问题的竞争条件。一旦比较/交换函数成功了，结点就被插入链表并可以被使用了。这里没有使用锁，因此不会产生死锁，并且 push()函数出色地实现了功能。

当然，现在已经有了在栈中增加数据的方法，还需要在栈中移出数据的方法。从表面上看，这要简单一些。

1 读取 head 当前的值。
2 读取 head->next。
3 将 head->next 设置为 head。
4 返回检索到的结点的值。
5 删除检索到的结点。

然而，在存在多个线程的情况下，这个问题就不这么简单了。如果同时有两个线程从栈中移出元素，他们可能在第 1 步中同时读取了相同的 head 值。如果一个线程在其他线程执行第 2 步前顺利执行到第 5 步，那么第二个线程将被解引用悬挂指针。这是写无锁代码中最大的问题之一。因此从现在开始，先忽略第 5 步并且先不删除结点。

但是，这也没有解决掉所有的问题。这里存在着另一个问题，如果两个线程读取同一个 head 值，那么它们将会返回同一个结点。这就违背了栈数据结构的目的，因此必须避免发生这种情况。你可以用 push() 中使用的方法来解决竞争，使用比较/交换来更新 head。如果比较/交换失败了，要么是因为在栈中插入了一个新结点，要么是因为另一个线程从栈中移出了你打算移出的结点。无论是哪一种情况，都需要返回到第 1 步（尽管比较/交换调用可以重新读取 head）。

一旦比较/交换调用成功了，那么这就是唯一的从栈中移出指定结点的线程。因此可以安全地执行第 4 步。pop() 如下所示。

```
template<typename T>
class lock_free_stack
{
public:
    void pop(T& result)
    {
        node* old_head=head.load();
        while(!head.compare_exchange_weak(old_head,old_head->next));
        result=old_head->data;
    }
};
```

尽管这种方法很好很简明，但是除了未删除的结点外还有一些别的问题。首先，当链表为空时它就行不通了，如果 head 是空指针，那么当它试图读取 next 指针时就会导致未定义的行为。这也很容易通过在 while 循环中检查空指针来解决。可以同时在空栈上引发异常或者返回一个 bool 来表明成功或失败。

第二个问题就是异常安全问题。当我们在第 3 章中首次介绍线程安全栈时，你可以看出只通过值返回对象会留下一个异常安全问题，当复制返回值的时候，如果引发了异常，那么此值就会被丢失。在这种情况下，传递对值的引用是一种解决方法。因为如果抛出异常的话，这可以确保栈不会被修改。可惜，这里不能使用这种方法。一旦你知道这是唯一一个返回结点的线程，你就可以安全地复制数据了。这就意味着这个结点已经

被移出队列了。因此，通过引用传递返回值的对象就不再是一个优势了，当然也可能只是返回值。如果想安全地返回值，就必须使用第 3 章中提到的另一个方法，返回一个指向数据值的（智能）指针。

如果返回智能指针，那么就可以只返回空指针来表明没有返回值，但是这就要求数据是在堆上被分配的。如果将堆分配作为 pop() 的一部分，你仍然没有做得更好，因为堆分配也可能会引发异常。反而，当把数据 push() 进栈时，可以为此数据分配内存——反正都要为结点分配内存。返回 std::shared_ptr<>不会引发异常，因此pop()是安全的。将这些总结起来得到清单 7.3 所示的代码。

**清单 7.3　缺少结点的无锁栈**

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;                       <-- ❶ data 现在由指针持有
        node* next;

        node(T const& data_):
            data(std::make_shared<T>(data_))           <-- ❷ 为新分配的 T 创建 std::shared_ptr
        {}
    };

    std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data);
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {                                                   ❸ 在解引用之前检查 old_head 不是一个空指针
        node* old_head=head.load();
        while(old_head &&                              <--
            !head.compare_exchange_weak(old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();   <-- ❹
    }
};
```

数据现在由指针持有 ❶，因此需要在结点构造函数中在堆上分配数据 ❷。在 compare_exchange_weak()循环中解引用 old_head 前，必须检查空指针。最后，如果存在与结点相关的值，那么就返回该值，否则就返回一个空指针。注意，尽管这是无锁的，但是它不是无等待的，因为在 push()和 pop()的 while 循环中，如果 compare_exchange_weak()一直失败的话，理论上可以一直循环下去。

如果有垃圾回收器在你后面打点（比如在 C#或 Java 这样的托管语言中），那么此工作就已完成了。一旦没有线程存取此结点，那么旧的结点被收集并被再次利用。然而，没有多少 C++编译器有垃圾回收器，因此通常需要自己整理。

## 7.2.2 停止恼人的泄漏：在无锁数据结构中管理内存

我们首先观察 pop()，当一个线程删除结点，而另一个线程仍然持有指向此结点的指针时，我们选择泄漏结点来避免竞争条件，那么就只能解引用了。尽管如此，在任何合理的 C++程序中，泄漏内存都是不可接受的。因此，我们必须做一些事。现在该考虑这个问题并且找出解决办法了。

最基本的问题就是，你想释放一个结点，但是直到你确保没有别的线程持有指向此结点的指针的时候，你才能释放此结点。如果只有一个线程曾经在一个特定的栈实例上调用 pop()，那么可以自由释放此结点。一旦结点被添加到栈中，push()就不会再操作此结点了。因此调用 pop()的线程就一定是唯一一个操作此结点的线程，并且可以安全地删除此结点。

另一方面，如果需要处理多个线程在同一个栈实例上调用 pop()的情况，那么就需要一些方法来追踪何时可以安全地删除结点。这就从根本上意味着你需要为结点写一个专用的垃圾回收器。现在，这可能听上去很可怕，尽管它确实很讨厌，但是也不是太糟糕。只需要检查结点，并且只检查在 pop()中存取的结点。不需要担心在 push()中存取的结点，因为它们只能被一个线程存取，直到它们在栈上为止。然而，多个线程可能在 pop()中存取同一个结点。

如果没有线程调用 pop()，那么可以删除目前等待删除的所有结点。因此，当你获得数据时，如果将此结点添加到“将被删除”的列表中，那么当没有线程调用 pop()时就可以删除它。如何知道有没有别的线程在调用 pop()呢？有个简单的方法——数清数目。如果在进入的时候计数器加一，在离开的时候计数器减一。那么当计数器为零的时候，就可以安全地删除“将被删除”列表中的结点。当然，此计数器必须为原子计数器，从而可以安全地被多个线程存取。清单 7.4 给出了修改后的 pop()函数，并且清单 7.5 列出了此实现的支撑函数。

**清单 7.4　当 pop()中没有线程时回收结点**

```
template<typename T>
class lock_free_stack
{
private:
    std::atomic<unsigned> threads_in_pop;      ❶ 原子变量
    void try_reclaim(node* old_head);
public:
    std::shared_ptr<T> pop()
```

```
    {
        ++threads_in_pop;                                    ❷ 在做任何其他事情前增
        node* old_head=head.load();                            加计数
        while(old_head &&
              !head.compare_exchange_weak(old_head,old_head->next));
        std::shared_ptr<T> res;
        if(old_head)
        {                                                   ❸ 如果可能，回收删除的
            res.swap(old_head->data);                         结点
        }
        try_reclaim(old_head);                              ❹ 从结点中提取数据，而不是
        return res;                                           复制指针
    }
};
```

原子变量 threads_in_pop❶被用作计数目前有多少线程试图从栈中移出数据项。在 pop() 开始的地方❷增加计数器，在 try_reclaim() 中减少计数器，而一旦结点被移出的时候就会调用此函数 ❹。因为可能将延迟删除结点，因此可以使用 swap() 来将数据从结点中删除 ❸，而不是仅仅复制指针。因此当不再需要时可以自动地删除此数据，而不会因为存在对未删除结点的引用而一直保持它。清单 7.5 给出了 try_reclaim() 的内部实现。

**清单 7.5 引用计数的回收机制**

```
template<typename T>
class lock_free_stack
{
private:
    std::atomic<node*> to_be_deleted;
    static void delete_nodes(node* nodes)
    {
        while(nodes)
        {
            node* next=nodes->next;
            delete nodes;
            nodes=next;
        }
    }                                                  列出将要被删除的结点 ❷
    void try_reclaim(node* old_head)                   清单
    {
        if(threads_in_pop==1)                   ❶
        {
            node* nodes_to_delete=to_be_deleted.exchange(nullptr);
            if(!--threads_in_pop)
            {                                          ❸ 是 pop()中唯一的线程吗?
                delete_nodes(nodes_to_delete);                    ❹
            }
            else if(nodes_to_delete)            ❺
            {
```

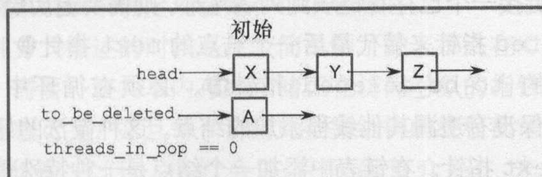
```
                chain_pending_nodes(nodes_to_delete);   ⑥
            }
            delete old_head;   ⑦
        }
        else
        {
            chain_pending_node(old_head);   ⑧
            --threads_in_pop;
        }
    }
    void chain_pending_nodes(node* nodes)
    {
        node* last=nodes;                          ⑨ 跟随下一个指针，链至
        while(node* const next=last->next)            末尾
        {
            last=next;
        }
        chain_pending_nodes(nodes,last);
    }
    void chain_pending_nodes(node* first,node* last)
    {
        last->next=to_be_deleted;   ⑩
        while(!to_be_deleted.compare_exchange_weak(
                  last->next,first));
    }                                              ⑪ 循环以保证 last->next
    void chain_pending_node(node* n)                  正确
    {
        chain_pending_nodes(n,n);   ⑫
    }
};
```

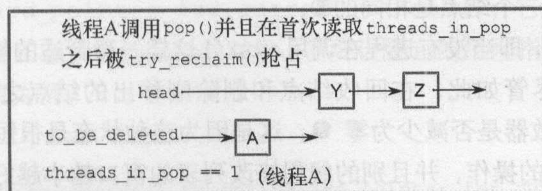
当回收结点时，如果 threads_in_pop 的值为 1❶，那么在 pop() 中这就是唯一的线程，这就意味着可以安全删除刚移动出来的结点❼，并且可能安全地删除等待的结点。如果计数器的值不为 1，那么删除任何结点都不安全，因此将此结点加入到等待的列表中❽。

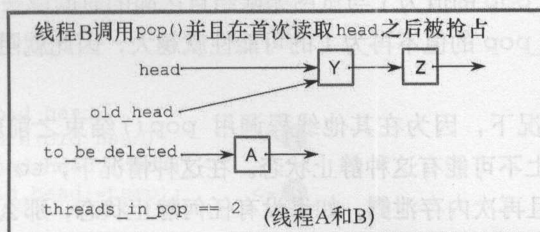
现在假设 threads_in_pop 的值为 1。此时需要回收等待的结点；如果不回收，那么这些结点将一直等待直到栈被销毁。回收结点时，首先用原子操作 exchange 来查找列表❷，然后将 threads_in_pop 的计数减一❸。如果计数减一后值为零，就可以得知没有别的线程存取此等待结点列表。可能会有新的等待结点，但是只要回收列表是安全的，就不需要操心这些新的等待结点。然后，调用 delete_nodes 来迭代此列表，并且删除结点❹。

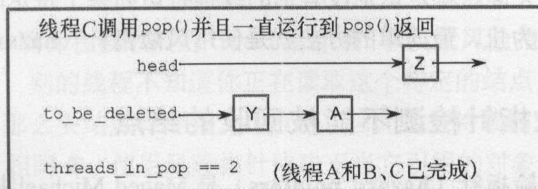
如果计数减一后值不为零，那么回收结点就不安全了。因此如果此时有等待的结点❺，那么就将此结点插入到等待删除结点列表的尾部❻。当多个线程同时存取数据结构时就有可能发生这种情况。别的线程可能在第一次取 threads_in_pop 值❶和查找列表❷之间调用 pop()。这就可能在列表中增加新的结点，并且此结点被一个或多个线程存取。在图 7.1 中，线程 C 增加结点 Y 至 to_be_deleted 列表中，即使线程 B 仍然引用结点 Y 作为 old_head，并且会读取此结点的 next 指针。因此线程 A 在删

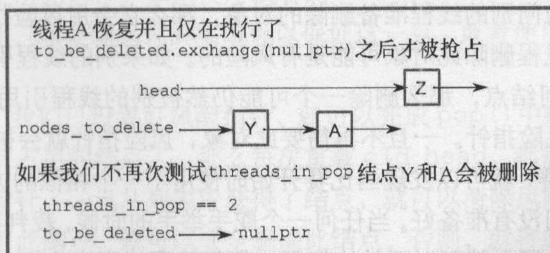
除结点的时候不可避免地会造成线程 B 未定义的行为。



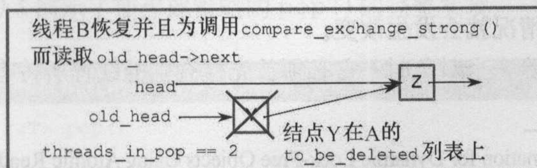




图 7.1 三个线程并发调用 pop()，在回收 try_reclaim()中删除的结点之后必须检查 threads_in_pop。

为了将等待删除的结点链接到等待列表中，需要重新使用结点的 next 指针来将它们链接起来。对于重新链接一个已存在链表到列表尾部，则需要遍历链表来找到尾部❾，用当前的 to_be_deleted 指针来替代最后一个结点的 next 指针❿，并且存储链表中的第一个结点作为新的 to_be_deleted 指针⓫。必须在循环中使用 compare_exchange_weak 来确保没有遗漏其他线程添加的结点。这种做法的好处是当链表发生变化时，从链尾更新 next 指针。在链表中添加一个结点是一种特殊情况，即链表中添加的第一个结点与最后一个结点是相同的⓬。

在低负载的情况下，即当没有进程在调用 pop()这样一种合适的静态点的时候，这种方法是很有效的。尽管如此，在回收结点和删除刚移出的结点之前❼都需要检查 threads_in_pop 计数器是否减少为零 ❸，这是因为这种状态是很短暂的。删除结点是一种会消耗一定时间的操作，并且别的线程修改列表的窗口越小越好。在线程第一次发现 threads_in_pop 的值为 1 与试图删除结点之间的时间越长，别的线程调用 pop()以及 threads_in_pop 的值不再为 1 的可能性就越大，因此就阻止了此结点被真正的删除。

在高负载的情况下，因为在其他线程调用 pop()结束之前就会有别的线程调用 pop()，因此基本上不可能有这种静止状态。在这种情况下，to_be_deleted 列表很容易就越界了，并且再次内存泄露。如果没有任何静止状态，那么就需要用别的方法来回收结点。关键点就是识别没有别的线程将访问某个特定的结点，那么就可以回收此结点了。迄今为止，最简单的方法就是使用**风险指针（hazard pointers）**。

### 7.2.3 用风险指针检测不能被回收的结点

术语**风险指针（hazard pointers）**是 Maged Michael[1]提出的一种技术。基本思想就是如果一个线程准备访问别的线程准备删除的对象，那么它会用风险指针来引用对象，因此就可以通知别的线程删除此对象可能是有风险的。如果别的线程引用此结点，并且准备通过此引用来访问结点，那么删除一个可能仍然被别的线程引用的结点是有危险的，因此它们被称为风险指针。一旦不再需要此对象，风险指针就会被清除了。如果你看过牛津/剑桥划船比赛，就可以发现当比赛开始时使用了一个相似的方法：每艘船的舵手都可以举手示意他们没有准备好。当任何一个舵手举手的时候，裁判都不能开始比赛。如果舵手都没有举手，那么就可以开始比赛了。但是只要比赛尚未开始，任何一个舵手都可以举手。此时情况就会发生改变。

当线程试图删除一个对象时，它必须首先检查别的线程所持有的风险指针。如果没

---

[1] "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes," Maged M. Michael, in *PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*(2002), ISBN 1-58113-485-1

有风险指针引用此对象，那么就可以删除此对象。否则，它必须之后才能被处理。周期性地检查对象列表来确定现在是否可以删除它。

用这种方式描述的时候是比较简单明了的，那么在C++中如何实现呢？

首先，需要一块共享内存来存储正在访问对象的指针，即风险指针本身。此地址必须对所有线程可见，并且每个访问此数据结构的线程都需要其中一段内存。如何正确并且有效地分配它们，我们会在后面章节介绍。先假设已经有这样一个函数 get_hazard_pointer_for_current_thread()，它返回风险指针的引用。当线程试图解引用正在读取的指针的时候，需要设置它的风险指针。在这里我们以解引用列表的 head 值为例。

```
std::shared_ptr<T> pop()
{
    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();          <-❶
    node* temp;
    do
    {
        temp=old_head;
        hp.store(old_head);              <-❷
        old_head=head.load();
    } while(old_head!=temp);             <-❸
    // ...
}
```

在 while 循环中，确保在读取旧的 head 指针 ❶ 和设置风险指针 ❷ 之间，结点未被删除。在此窗口内，别的线程不知道你正在读取这个特定的结点。幸运的是，如果旧的头指针将被删除，那么头结点肯定会发生改变，因此必须持续循环直到确定头指针与之前设置的风险指针相同 ❸。使用风险指针取决于当它引用的对象被删除后，仍然可以安全地使用此指针。如果使用缺省的 new 和 delete 实现，那么就会导致未定义的行为。因此，需要确保你的实现可以保证这一点，或者使用允许这种行为的自定义分配器。

现在，我们已设置好风险指针，就可以完成 pop() 中剩余的代码。此时没有线程将会删除你所占用的结点了。那么每次重载 old_head，都需要在解引用新读取的指针前更新风险指针。一旦从列表中获得了结点，就可以清除风险指针。如果此时没有别的风险指针引用此结点，就可以安全删除此结点。否则，就将此结点加入等待稍后删除的结点列表。清单 7.6 演示了使用此策略的 pop() 的完整实现。

**清单 7.6　使用风险指针的 pop()实现**

```
std::shared_ptr<T> pop()
{
  std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
  node* old_head=head.load();
```

```
    do
    {
        node* temp;                          ❶ 一直循环到你将风险指
        do                                      针设置到 head 上
        {
            temp=old_head;
            hp.store(old_head);
            old_head=head.load();
        } while(old_head!=temp);
    }
    while(old_head &&
          !head.compare_exchange_strong(old_head,old_head->next));
    hp.store(nullptr);                       ❷ 当你完成时清除风险
    std::shared_ptr<T> res;                     指针
    if(old_head)                                          ❸ 在你删除一个结点前
    {                                                       检查风险指针是否引
        res.swap(old_head->data);                           用它
        if(outstanding_hazard_pointers_for(old_head))
        {
            reclaim_later(old_head);         ❹
        }
        else
        {
            delete old_head;                 ❺
        }
        delete_nodes_with_no_hazards();      ❻
    }
    return res;
}
```

首先，将设置风险指针放到外部循环中，如果比较/交换失败，则重载 old_head❶。这里使用 compare_exchange_strong()是因为在这个 while 循环中确实有效，compare_exchange_weak()中虚假的错误会导致不必要地重置风险指针。这就确保了在解引用 old_head 前设置了正确的风险指针。一旦声明此结点是你的，就可以清除你的风险指针❷。如果你得到一个结点，就需要检查别的线程拥有的风险指针是否引用它❸。如果存在这样的风险指针，那么必须将它放入到稍后回收的列表中❹。否则，就可以立刻删除它❺。最后，调用 reclaim_late()来检查所有结点。如果没有别的风险指针引用这些结点，那么可以安全删除它们❻。任何有风险指针的结点都将留待下一个线程调用 pop()。

当然，在这里有很多新函数——get_hazard_pointer_for_current_thread()、reclaim_later()、outstanding_hazard_pointers_for()，以及 delete_nodes_with_no_hazards()中有很多细节部分——让我们来了解这些函数并且看看它们是如何工作的。

调用 get_hazard_pointer_for_current_thread()给线程分配风险指针的具体机制并不影响程序的逻辑（稍后就可以看到对效率有影响）。因此我们先用一个简

单的结构来实现，一个固定大小数组存放线程 ID 和指针对。`get_hazard_pointer_for_current_thread()` 检索整个数组来寻找第一个空闲的位置，并且将此位置的 ID 值设为当前线程的 ID。当线程退出时，此位置的 ID 值被重置为默认值 `std::thread::id()`，从而此位置就被释放出来了，如清单 7.7 所示。

**清单 7.7 get_hazard_pointer_for_current_thread()的简单实现**

```
unsigned const max_hazard_pointers=100;
struct hazard_pointer
{
    std::atomic<std::thread::id> id;
    std::atomic<void*> pointer;
};
hazard_pointer hazard_pointers[max_hazard_pointers];

class hp_owner
{
    hazard_pointer* hp;

public:
    hp_owner(hp_owner const&)=delete;
    hp_owner operator=(hp_owner const&)=delete;
    hp_owner():
        hp(nullptr)
    {
        for(unsigned i=0;i<max_hazard_pointers;++i)
        {
            std::thread::id old_id;
            if(hazard_pointers[i].id.compare_exchange_strong(   // 试着获取风险指针的所有权
                old_id,std::this_thread::get_id()))
            {
                hp=&hazard_pointers[i];
                break;
            }
        }
        if(!hp)    // 1
        {
            throw std::runtime_error("No hazard pointers available");
        }
    }

    std::atomic<void*>& get_pointer()
    {
        return hp->pointer;
    }

    ~hp_owner()    // 2
    {
        hp->pointer.store(nullptr);
        hp->id.store(std::thread::id());
    }
};
```

```
std::atomic<void*>& get_hazard_pointer_for_current_thread()   <-❸
{
    thread_local static hp_owner hazard;   <-                 ❹ 每个线程有自己的风险
    return hazard.get_pointer();   <-❺                          指针
}
```

get_hazard_pointer_for_current_thread()的实现看似简单，其实不然 ❸：它用 hp_owner❹ 类型的 thread_local 变量来存储当前线程的风险指针。然后它返回此对象的指针 ❺。它的原理如下：每个线程第一次调用此函数的时候，创建一个新的 hp_owner 实例。这个新实例构造器搜索所有者/指针对的表格来寻找一个值，此值没有所有者。它使用 compare_exchange_strong()来检查没有所有者的值并且获得它 ❷。如果 compare_exchange_strong()失败了，那么就说明另一个线程拥有此值，那么就需要去检查下一个值。如果 compare_exchange_strong()成功了，那么当前线程就成功获得此值。此时就存储此值，并且停止检索 ❸。如果在整个列表中都没有找到一个空闲的值 ❹，那么就表示有太多线程使用了风险指针，此时就抛出异常。

一旦为一个给定的线程创造了 hp_owner 实例，那么之后的存取就变得更快了，因为缓存了指针，就不需要再次扫描表格了。

当每个线程退出时，为该线程创造的 hp_owner 实例就被销毁了。析构函数在设置所有者的 ID 的值为 std::thread::id()前将指针的值重置为 nullptr，这样稍后别的线程就可以重新使用此值 ❺。

用这种方式实现 get_hazard_pointer_for_current_thread()，那么 outstanding_hazard_pointer_for_current_thread()的实现就变得简单了，只需要检索整个风险指针表来寻找这个位置。

```
bool outstanding_hazard_pointers_for(void* p)
{
    for(unsigned i=0;i<max_hazard_pointers;++i)
    {
        if(hazard_pointers[i].pointer.load()==p)
        {
            return true;
        }
    }
    return false;
}
```

现在甚至都不需要检索表来得知每个位置是否拥有所有者，没有所有者的位置将会有一个空指针，因此这个比较函数将返回 false，这样就简化了代码。

在简单链表中，reclaim_later()和 delete_nodes_with_no_hazards()可以工作，reclaim_later()只添加结点至列表中，delete_nodes_with_no_hazards()扫描整个列表，删除没有风险的值。清单 7.8 就是一个实现。

**清单 7.8 回收函数的简单实现**

```
template<typename T>
void do_delete(void* p)
{
    delete static_cast<T*>(p);
}

struct data_to_reclaim
{
    void* data;
    std::function<void(void*)> deleter;
    data_to_reclaim* next;

    template<typename T>
    data_to_reclaim(T* p):    <--❶
        data(p),
        deleter(&do_delete<T>),
        next(0)
    {}

    ~data_to_reclaim()
    {
        deleter(data);    <--❷
    }
};

std::atomic<data_to_reclaim*> nodes_to_reclaim;
void add_to_reclaim_list(data_to_reclaim* node)    <--❸
{
    node->next=nodes_to_reclaim.load();
    while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
}

template<typename T>
void reclaim_later(T* data)    <--❹
{
    add_to_reclaim_list(new data_to_reclaim(data));    <--❺
}

void delete_nodes_with_no_hazards()
{
    data_to_reclaim* current=nodes_to_reclaim.exchange(nullptr);    <--❻
    while(current)
    {
        data_to_reclaim* const next=current->next;
        if(!outstanding_hazard_pointers_for(current->data))    <--❼
        {
            delete current;    <--❽
        }
        else
        {
            add_to_reclaim_list(current);    <--❾
        }
        current=next;
    }
}
```

首先，我希望你发现 reclaim_later() 不是一个普通的函数，而是一个函数模板 ❹。这是因为风险指针是一种通用的工具，因此不希望绑定到具体的结点。你已经使用 std::atomic<void*>来存储指针了。因此需要处理任何指针类型，但是不能使用 void*类型。因为当你删除数据项的时候，delete 函数需要指针的实际类型。date_to_reclaim 的构造器可以很好地处理这个问题，就如以下所示。reclaim_later()只需要为你的指针生成一个新的 date_to_reclaim 实例，并且将它加入到回收列表中 ❺。add_to_reclaim_list() 本身 ❸ 是一个基于列表头结点的简单 compare_exchange_weak()循环。

因此，回到 data_to_reclaim 的构造函数 ❶，这个构造函数也是一个模板。它将被删除数据存储为 data 成员的 void*类型。然后存储指向 do_delete()实例的指针。do_delete()是一个简单的函数，将提供的 void*类型确定为选好的指针类型，然后删除它所指向的对象。std::function<>可以安全地实现这个函数指针，因此 data_to_reclaim 的析构函数可以调用存储的函数来删除数据 ❷。

当你在列表中增加结点时，不会调用 data_to_reclaim 的析构器。当没有风险指针指向此结点时就会调用此析构函数。这是 delete_nodes_with_no_hazards()的责任。

delete_nodes_with_no_hazards()首先用一个简单的 exchange()来声明所有将被回收的结点列表 ❻。这一简单但是关键的步骤确保了这是将回收这个结点集合的唯一线程。别的线程可以自由向列表中增加结点或者试图回收它们，并且不会影响此线程的操作。

然后，只要列表中仍然有结点，就轮流检查每个结点来看是否存在风险指针❼。如果没有，则安全删除此位置的值（即清除了存储的数据）❽。否则，就将此项增加到稍后回收列表中❾。

尽管这种简单实现能够安全回收删除的结点，但是它会增加很多处理难度。扫描风险指针数组需要检查 max_hazard_pointers 原子变量，并且每次调用 pop( )的时候都会执行这个操作。原子操作必定是很慢的——通常在计算机 CPU 上运行时会比实现同样效果的非原子操作慢 100 倍——这就使得 pop()变成很耗资源的操作。不仅需要扫描将要删除结点的风险指针列表，而且需要扫描等待列表中每个结点的风险指针列表。这当然不是个好主意。如果列表中有 max_hazard_pointers 个结点，那么就得扫描这些结点存储的风险指针。天啊！必须找到一种更好的方法。

### 使用风险指针的更佳的回收策略

当然，有更好的方法。这里我将介绍一种简单的风险指针实现来解释这种机制。第一件事就是用内存资源换取效率。你不再试图回收任何结点除非表中的结点数多于 max_hazard_pointers，而不再每次都调用 pop()来检查回收列表中每个结点。用

这种方式可以保证至少回收一个结点。如果只是等到表中有 max_hazard_pointers+1 个结点，这种方法也没有更好。一旦你得到 max_hazard_pointers 个结点，就开始调用 pop() 来回收结点，这种方法也没有更好。但是如果你等到表中有 2*max_hazard_pointers 个结点，就可以确保收回至少 max_hazard_pointers 个结点，并且在你回收任何结点前至少会 max_hazard_pointers 次调用 pop()。这种方法就比较好了。你在 max_hazard_pointers 次调用 pop() 的时候都会检查 2*max_hazard_pointers 个结点，并且至少回收 max_hazard_pointers 个结点。而不需要每次调用 push() 的时候检查 max_hazard_pointers 个结点。这样是很有效的，每次调用 pop() 都会检查两个结点，回收一个结点。

这种方案也有缺点（除了增加内存使用），需要计数回收列表中的结点，这就意味着要使用原子计数，并且多个线程还在竞争访问此回收列表。如果有共享内存，就可以用增加的内存使用换取一个更好的回收策略。每个线程在线程本地变量上有自己的回收列表。因此就不需要用来计数的原子变量以及存取列表。相对的，就分配了 max_hazard_pointers*max_hazard_pointers 个结点。如果线程在回收完它所有的结点前退出了，它们就可以像以前一样存储在全局列表中，并且加入到下一个执行回收操作的线程的本地列表中。

风险指针的另一个缺点是他们涉及到 IBM 提交的专利申请[1]。如果在一个承认此专利有效的国家写软件，那么就需要确保获得一个合适的许可。一些无锁内存回收机制可以共用此技术。这是一个很活跃的研究领域，很多公司都在竭尽所能地提交专利申请。你可能会提出这样的疑问，为什么我花了这么多篇幅介绍一种很多人都不能使用的一种技术，这是一个合理的问题。首先，有可能在不获得许可的情况下使用该技术。例如，如果你在 GPL[2]下开发免费软件，你的软件可以被 IBM 的非不主张条款[3]所覆盖。就可以使用该技术。第二，更重要的一点是，对这项技术的解释展示了在写无锁代码的时候需要考虑哪些重要的事情，如原子操作的开销。

因此，是否存在可以用在无锁代码的非专利内存技术？幸运的是，确实有。一种技术就是引用计数。

### 7.2.4 使用引用计数检测结点

回顾 7.2.2 节，删除结点的问题就在于检测哪些结点正在被别的线程读取。如果可

---

[1] Maged M. Michael, U.S. Patent and Trademark Office application number 20040107227, “Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation.”

[2] GNU General Public License http://www.gnu.org/licenses/gpl.html

[3] IBM Statement of Non-Assertion of Named Patents Against OSS, http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf

以精确识别出哪些结点正在被引用以及何时没有线程读取这些结点，那么就可以删除此结点。风险指针通过存储读取每个结点的线程数来处理此问题。引用技术通过存储一定数量的线程读取结点来处理这个问题。

这种方法看上去更好更直接，但是在实际中很难处理。首先，你可能认为std::shared_ptr<>可以处理这种问题；毕竟，这是一个引用计数指针。不幸的是，尽管std::shared_ptr<>中的一些操作是原子的，但是它们不能保证是无锁的。尽管这与原子类型上的任何操作并没有不同，但是在许多情况下std::shared_ptr<>被使用，并且使得原子操作是无锁的会导致使用这个类有花费。如果你的平台提供这样一个实现，即当std::atomic_is_lock_free(&some_shared_ptr)返回true，所有的内存回收事件都离开。如清单 7.9 所示，其中只使用 std::shared_ptr <node>。

**清单 7.9 使用无锁的 std::shared_ptr<>的无锁栈实现**

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::shared_ptr<node> next;
        node(T const& data_):
            data(std::make_shared<T>(data_))
        {}
    };
    std::shared_ptr<node> head;
public:
    void push(T const& data)
    {
        std::shared_ptr<node> const new_node=std::make_shared<node>(data);
        new_node->next=head.load();
        while(!std::atomic_compare_exchange_weak(&head,
                &new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        std::shared_ptr<node> old_head=std::atomic_load(&head);
        while(old_head && !std::atomic_compare_exchange_weak(&head,
                &old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();
    }
};
```

在可能的情况下，std::shared_ptr<>实现不是无锁的，需要手工处理引用计数。一种可能的技术涉及为每个结点使用不止一个而是两个引用计数，一个内部计数和一个外部计数。这两个计数值之和是结点总的引用数。外部计数始终与结点指针在一起，

并且每次读取指针的时候外部计数增一。当读取结点结束时，内部计数减一。读取指针这样一个简单操作会导致外部计数增一，并且在此操作结束时内部计数减一。

当内部计数/指针对不在需要时（即多个线程不再访问结点时），内部计数增加外部计数的值减一，并废除外部计数。一旦内部计数的值为零，就没有引用结点，此时可删除此结点。使用原子操作来更新共享数据也是很重要的。现在我们来看一个使用这种技术来确保结点只会被安全收回的无锁栈的实现。

更好的内部数据结构和push()的实现如清单 7.10 所示。

**清单 7.10 在使用两个引用计数的无锁栈中入栈结点**

```
template<typename T>
class lock_free_stack
{
private:
    struct node;
    struct counted_node_ptr         <-❶
    {
        int external_count;
        node* ptr;
    };

    struct node
    {
        std::shared_ptr<T> data;
        std::atomic<int> internal_count;   <-❷
        counted_node_ptr next;     <-❸

        node(T const& data_):
            data(std::make_shared<T>(data_)),
            internal_count(0)
        {}
    };

    std::atomic<counted_node_ptr> head;   <-❹
public:
    ~lock_free_stack()
    {
        while(pop());
    }

    void push(T const& data)     <-❺
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        new_node.ptr->next=head.load();
        while(!head.compare_exchange_weak(new_node.ptr->next,new_node));
    }
};
```

首先，在 counted_node_ptr 结构中包含了外部变量与结点的指针 ❶。在 node 结构体❸中将使用 counted_node_ptr 类型的 next 指针以及内部变量 ❷。因为 counted_node_ptr 是一种简单的结构，因此在 std::atomic<>模板中使用它作为列表的头结点❹。

在这些支持双字比较和交换操作的平台上，这个结构足够小，使得 std::atomic<counted_node_ptr>是无锁的。如果不是在你的平台上，那么最好使用清单 7.9 中提到的 std::shared_ptr<>，因为当类型太大使得平台的原子指令不能实现时，std::atomic<>将使用一个互斥元来保证原子性（因此最后使得你的“无锁”算法变成基于锁的算法）。或者，如果你想限制计数的位数，并且你知道你的平台中指针有空闲位（例如，地址空间只有 48 位但是指针有 64 位），你可以在单个字中将计数存储在指针的空闲位中。这种方法需要与平台相关的知识，这就超出了本书的范围了。

push()相对简单一些❺。构造一个指向新分配结点的 counted_node_ptr，并将它的 next 赋值为当前的 head。之后用 compare_exchange_weak()来给 head 赋值，就像之前的清单所示。设置计数器时，将内部计数设为零，外部计数设为一。因为这是新创建的结点，只有一个外部引用（即 head 本身）。

同理，pop()的实现也复杂了一些，如清单 7.11 所示。

**清单 7.11 使用两个引用计数从无锁栈中出栈一个结点**

```
template<typename T>
class lock_free_stack
{
private:
    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter));  <-❶

        old_counter.external_count=new_counter.external_count;
    }

public:
    std::shared_ptr<T> pop()#
    {
        counted_node_ptr old_head=head.load();
        for(;;)
        {
            increase_head_count(old_head);
            node* const ptr=old_head.ptr;    <-❷
            if(!ptr)
```

```
            {
                return std::shared_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next))      <-3
            {
                std::shared_ptr<T> res;
                res.swap(ptr->data);          <-4
                int const count_increase=old_head.external_count-2;     <-5
                if(ptr->internal_count.fetch_add(count_increase)==     <-6
                    -count_increase)
                {
                    delete ptr;
                }
                return res;       <-7
            }
            else if(ptr->internal_count.fetch_sub(1)==1)
            {
                delete ptr;      <-8
            }
        }
    }
};
```

这里，一旦你载入了 head 的值，就必须将引用此 head 结点的外部计数的值增一，用以表明你引用了此结点并且确保解引用是安全的。如果在增加引用计数前解引用此指针，那么另一个线程就可以在你读取这个结点前释放该结点，因此使得它变成悬挂指针。这是使用两个分开的引用计数的主要原因。通过增加外部引用计数，就可以保证直到你访问时指针仍然是有效的。在 compare_exchange_strong() 循环内部增加计数的值 ❶，确保了没有别的线程在此时改变它。

一旦增加了计数，为了访问它指向的结点，可以安全解引用从 head 载入的 ptr 的值 ❷。如果指针为空，表明位于链表的尾部没有位置了。如果指针不为空，就可以通过在 head 上调用 compare_exchange_strong() 来移动结点 ❸。

如果 compare_exchange_strong() 成功了，就可以拥有该结点，并且交换出 data 以备以后返回它 ❹。这就确保了就算别的线程读取栈的时候一直持有指向此结点的指针，data 也不需要一直保持。然后就可以使用原子操作 fetch_add 将结点外部计数的值加到内部计数上 ❻。如果当前引用计数的值为零，那么先前你增加的值（即 fetch_add 的返回值）就是负数，此时就可以删除这个结点。请注意你增加的值比外部计数的值减少 2❺。你已经从列表中移出了结点，因此计数减一，并且这个线程不在读取这个结点，因此计数的值再次减一。无论是否删除此结点，程序都结束了，因此可以返回 data❼。

如果比较/交换 ❸ 失败了，则表明在此之前另一个线程移动了该结点，或者另一个线程入栈了一个新结点。不管怎样，你都需要用比较/交换返回的 head 新值重新开始。

但是首先你必须减少你试图移动的结点的引用计数。该线程不会再读取它了。如果这是持有引用的最后一个线程（因为另一个线程将它从栈中移出），那么内部引用计数的值为一，因此减少一将使得值变为零。在这种情况下，可以在循环之前删除此结点❽。

迄今为止，所有原子操作使用了默认的 std::memory_order_seq_cst 内存顺序。在大多数系统中，这种方法比别的内存顺序消耗更多的执行时间以及同步开销。现在，你有决定数据结构逻辑的权利，就可以考虑放松一些内存顺序要求。可以减少使用栈的不必要的开销。因此，先不考虑栈，考虑一下无锁队列的设计。检查栈操作并问问自己，对于一些操作是否可以使用更简单的内存顺序并且获得同样的安全性？

## 7.2.5 将内存模型应用至无锁栈

在改变内存顺序前，你需要检查操作以及它们之间的关系。然后就可以寻找提供这些关系的最小内存顺序。为了实现这一点，就必须在不同场景下从线程角度考虑情况。最简单的场景就是一个线程入栈一个数据项，并且稍后另一个线程将那个数据项出栈，我们先考虑这种情况。

在这种简单情况下，涉及数据的三个重要部分。第一部分是用来传输 head 数据的 counted_node_ptr。第二部分是 head 引用的结点数据结构。第三部分是结点指向的数据项。

线程 push() 的时候首先构造数据项和结点，然后设置 head。线程 pop() 的时候首先加载 head 的值，然后基于 head 做一个比较/交换循环来增加引用计数，最后读取结点数据结构来得到 next 的值。在这里可以看出这样一种关系，next 的值是一个普通的非原子性对象，因此为了安全读取它，必须存在存储（入栈线程执行的操作）发生在加载（出栈线程执行的操作）之前这样一种关系。因为 push() 中唯一的原子操作是 compare_exchange_weak()，所以需要一个释放操作来实现在线程间实现这样一种先后顺序关系，而且 compare_exchange_weak() 必须是 std::memory_order_release 或者更强的。如果 compare_exchange_weak() 失败了，那么就继续循环并且不做任何改变，因此在这种情况下需要使用 std::memory_order_relaxed。

```
void push(T const& data)
{
    counted_node_ptr new_node;
    new_node.ptr=new node(data);
    new_node.external_count=1;
    new_node.ptr->next=head.load(std::memory_order_relaxed)
    while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
        std::memory_order_release,std::memory_order_relaxed));
}
```

那么 pop() 的代码怎么样呢？为了实现这种先后顺序关系，你必须在读取 next 之前有一个 std::memory_order_acquire 或者更强的操作。解引用指针读取的 next

值是 increase_head_count()中的 compare_exchange_strong()读取的旧值。因此如果成功的话就需要有先后顺序。正如在 push()中一样，如果交换失败的话，只需要继续循环，因此失败时可以使用任意的顺序。

```
void increase_head_count(counted_node_ptr& old_counter)
{
    counted_node_ptr new_counter;
    do
    {
        new_counter=old_counter;
        ++new_counter.external_count;
    }
    while(!head.compare_exchange_strong(old_counter,new_counter,
        std::memory_order_acquire,std::memory_order_relaxed));
    old_counter.external_count=new_counter.external_count;
}
```

如果 compare_exchange_strong()调用成功，那么读取的结点的 ptr 值设置为 old_counter 中存储的值。因为 push()中的存储是一个释放操作，并且 compare_exchange_strong()是一个获取操作，因此存储与加载同步而且存在先后发生顺序的关系。所以，push()中存储 ptr 发生在 pop()中读取 ptr->next 之后，因此是安全的。

注意，在最初的 head.load()中内存顺序并不是很重要，因此可以安全使用 std::memory_order_relaxed。

下一步，compare_exchange_strong()将 head 的值设为 old_head.ptr->next。是否需要操作来确保线程的数据完整性？如果交换成功就读取 ptr->data，此时就需要确保在线程中 push()存储 ptr->data 的操作发生在线程加载它之前。尽管如此，你已经得到如下保证，increase_head_count()中的获取操作保证了 push()线程中的存储和比较/交换操作存在同步关系。因为 push()线程中存储 data 发生在存储 head 之前，调用 increase_head_count()发生在加载 ptr->data 之前，所以就存在一种先后顺序关系。并且即使 pop()中的比较/交换使用 std::memory_order_relaxed，这种先后顺序关系也是存在的。ptr->data 改变的唯一的地方就是调用 swap()，并且没有别的线程可以在同一个结点上进行操作，这就是整个比较/交换。

如果 compare_exchange_strong()失败了，直到下一次循环的时候才会访问 old_head 的新值。并且你决定了 increased_head_count()中的 std::memory_order_acquire 是足够的，因此 std::memory_order_relaxed 也是足够的。

那么其他线程呢？是否需要更强的方式来保证别的线程是安全的？答案是否定的。因为只有比较/交换操作会改变 head。因为这些是“读—修改—写”操作，它们通过 push()中的比较/交换形成了部分释放顺序。因此，push()中的 compare_exchange_weak()与调用 increase_head_count()中的 compare_exchange_strong()同步，它读取

存储的值，即使别的线程同时在修改 head。

因此，你基本上完成了，只需要处理修改引用计数的 fetch_add() 的操作。返回这个结点数据的线程可以继续，因为没有别的线程会修改这个结点数据。尽管如此，任何没有成功取值的线程知道别的线程的确修改了结点数据，它使用 swap() 获得引用的数据项。因此，为了避免数据竞争，你需要确保 swap() 发生在 delete 之前。实现它的一个简单方式就是在成功返回分支的 fetch_add() 中使用 std::memory_order_release，并且在再次循环分支的 fetch_add() 中使用 std::memory_order_acquire。还可以进一步简化设计，只有一个线程进行删除操作（将计数设置为零的线程），也只有此线程需要进行获取操作。因为 fetch_add() 是一个“读—修改—写”操作，它组成了释放顺序的一部分，因此可以使用额外的 load()。如果再次循环分支将引用计数减为零，那么为了保证同步关系可以使用 std::memory_order_acquire 来重载引用计数，并且 fetch_add() 可以使用 std::memory_order_relaxed。清单 7.12 所示就是使用新的 pop() 的栈的最终实现。

**清单 7.12　使用引用计数和放松原子操作的无锁栈**

```
template<typename T>
class lock_free_stack
{
private:
    struct node;

    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };

    struct node
    {
        std::shared_ptr<T> data;
        std::atomic<int> internal_count;
        counted_node_ptr next;

        node(T const& data_):
            data(std::make_shared<T>(data_)),
            internal_count(0)
        {}
    };

    std::atomic<counted_node_ptr> head;

    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
```

```
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter,
                                             std::memory_order_acquire,
                                             std::memory_order_relaxed));

        old_counter.external_count=new_counter.external_count;
    }
public:
    ~lock_free_stack()
    {
        while(pop());
    }
    void push(T const& data)
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        new_node.ptr->next=head.load(std::memory_order_relaxed)
        while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
                                           std::memory_order_release,
                                           std::memory_order_relaxed));
    }
    std::shared_ptr<T> pop()
    {
        counted_node_ptr old_head=
            head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_head_count(old_head);
            node* const ptr=old_head.ptr;
            if(!ptr)
            {
                return std::shared_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next,
                                             std::memory_order_relaxed))
            {
                std::shared_ptr<T> res;
                res.swap(ptr->data);

                int const count_increase=old_head.external_count-2;

                if(ptr->internal_count.fetch_add(count_increase,
                        std::memory_order_release)==-count_increase)
                {
                    delete ptr;
                }

                return res;
            }
            else if(ptr->internal_count.fetch_add(-1,
```

```
                std::memory_order_relaxed)==1)
            {
                ptr->internal_count.load(std::memory_order_acquire);
                delete ptr;
            }
        }
    }
};
```

这是一个实验，但是最后成功了。通过使用更放松的操作，在没有影响正确性的情况下提升了性能。正如你所见，这里的 pop() 实现有 37 行代码，在清单 6.1 基于锁的栈中 pop() 有 8 行代码，在清单 7.2 不使用内存管理的无锁栈中 pop() 有 7 行代码。现在我们考虑写一个无锁队列，你可以看到一个相似的模式，无锁代码中的很多复杂性都来自于管理内存。

### 7.2.6　编写不用锁的线程安全队列

队列与栈有所不同。因为队列中 push() 和 pop() 操作读取了数据结构的不同部分，而栈中这两个操作读取了相同的头结点。所以同步要求就不一样了。你需要确保一端所作出的改变能被另一端正确地读取。尽管如此，清单 6.6 中队列的 try_pop() 结构与清单 7.2 简单无锁栈中的 pop() 区别并不是很大，因此可以合理假设无锁代码不会不相似。

如果以清单 6.6 作为基础，就需要两个结点指针，一个指针指向 head，一个指针指向 tail。多个线程将会读取它们，为了去掉相关的互斥元，最好是原子操作。下面我们来做一些小的改变来看看效果如何。清单 7.13 展示了效果。

**清单 7.13　单生产者单消费者的无锁队列**

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;
        node():
            next(nullptr)
        {}
    };

    std::atomic<node*> head;
    std::atomic<node*> tail;
    node* pop_head()
    {
```

```
        node* const old_head=head.load();
        if(old_head==tail.load())       ←❶
        {
            return nullptr;
        }
        head.store(old_head->next);
        return old_head;
    }
public:
    lock_free_queue():
        head(new node),tail(head.load())
    {}

    lock_free_queue(const lock_free_queue& other)=delete;
    lock_free_queue& operator=(const lock_free_queue& other)=delete;

    ~lock_free_queue()
    {
        while(node* const old_head=head.load())
        {
            head.store(old_head->next);
            delete old_head;
        }
    }
    std::shared_ptr<T> pop()
    {
        node* old_head=pop_head();
        if(!old_head)
        {
            return std::shared_ptr<T>();
        }

        std::shared_ptr<T> const res(old_head->data);   ←❷
        delete old_head;
        return res;
    }

    void push(T new_value)
    {
        std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
        node* p=new node;                              ❸
        node* const old_tail=tail.load();    ←❹
        old_tail->data.swap(new_data);    ←❺
        old_tail->next=p;    ←❻
        tail.store(p);    ←❼
    }
};
```

看起来似乎没什么不好。如果一次只有一个线程调用 push()，并且只有一个线程调用 pop()，就工作的很好了。在这种情况下，重要的是 push() 和 pop() 的发生顺序关系以确保可以安全获取 data。tail 存储❼与 tail 加载 ❶ 同时发生；存储先前结点的 data 指针 ❺ 发生在存储 tail 之前；加载 tail 发生在加载 data 指针之前 ❷，因此存储 data 发生在加载之前，这就是安全的。这是一个性能良好的单生产者、单消

费者（**single-producer, single-consumer, SPSC**）队列。

当多个线程同时调用 push() 或多个线程同时调用 pop( )的时候就会存在问题。首先来看 push()。如两个线程同时调用 push()，它们都会分配新节点作为新的哑元结点❸，都会读取相同的 tail❹，并且设置 date 和 next 指针❺、❻时都会同时更新同一个结点的数据成员。这就是数据竞争！

pop_head() 中也存在类似的问题。如果两个线程同时调用 pop_head，就会读取同一个 head，并且会用同一个 next 指针覆盖旧值。这两个线程现在认为他们得到了相同的结点——这是有很大危害的。你不但要确保只有一个线程 pop() 结点，并且需要确保别的线程可以安全访问 head 的下一个结点。这就是无锁栈的 pop() 遇到的问题，因此很多方法可以用在这里。

如果 pop() 是一个"已解决的问题"，那么 push() 呢？问题就是为了得到 push() 和 pop() 的先后顺序关系，需要在更新 tail 前设置哑元结点的数据项。这就意味着同时调用 push() 会在这些相同的数据项上产生竞争，因为他们读取了相同的 tail 指针。

### 1. 处理 push()中的多个线程

一种选择是在真正的结点间增加一个哑元结点。这样，当前 tail 结点只需要更新它的 next 指针，因此可以是原子的。如果一个线程成功地将它的 next 指针从空改变为新结点，就代表它成功地增加了指针；否则，它就必须再次开始并且重新读取 tail。这就需要对稍微改变 pop() 来丢弃有空数据指针的结点，并且再次循环。缺点就是每次调用 pop( )都会移出两个结点，并且会有两倍内存分配。

第二种选择是使得 data 指针是原子的，并且调用比较/交换来设置它。如果调用成功，那么这就是 tail 结点，并且可以安全地将 next 指针设置为新结点，然后更新 tail。如果另一个线程已经存储了此数据，导致比较/交换失败了，那么就再次循环，重新读取 tail 然后重新开始。如果 std::shared_ptr<>的原子操作是无锁的，那么整个就是无锁的。如果不是，就需要别的方法。一种可能就是让 pop() 返回 std::unique_ptr<>（毕竟，这是对象的唯一引用）并且将数据用普通指针存储在队列里。这就允许你将它存储为 std::atomic<T*>，这样你就可以使用 compare_exchange_strong()。如果你使用清单 7.11 中的引用计数方法在多线程的情况下处理 pop()，那么 push() 就如清单 7.14 所示。

**清单 7.14 首次（很逊的）尝试修订 push()**

```
void push(T new_value)
{
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new node;
```

```
    new_next.external_count=1;
    for(;;)
    {
        node* const old_tail=tail.load();        ←❶
        T* old_data=nullptr;
        if(old_tail->data.compare_exchange_strong(
            old_data,new_data.get()))              ←❷
        {
            old_tail->next=new_next;
            tail.store(new_next.ptr);             ←❸
            new_data.release();
            break;
        }
    }
}
```

使用引用计数方法避免了特定的竞争，但是这不是 push() 中唯一的竞争。如果你看看清单 7.14 中 push() 的修订版本，就会发现栈中有这样一段代码：加载一个原子指针 ❶ 并且解引用那个指针 ❷。同时，另一个线程可以更新那个指针 ❸，最后指向再分配的结点（在 pop() 中）。如果在你解引用那个指针前再分配那个结点，就会产生不确定的行为。天哪！它试图像 head 一样在 tail 中增加一个外部计数，但是每个结点在队列先前的结点的 next 指针中已经有一个外部计数了。同一个结点拥有两个外部计数就意味着要需要修改引用计数方法来避免太早删除该结点。你可以这样处理，即在 node 结构体中计算外部计数的数量，并且当每个外部计数被销毁的时候（以及将相关的外部计数加到内部计数的时候）减少它的数量。如果结点的内部计数为零并且没有外部计数，此时就可以安全删除该结点。最初我是从 Joe Seigh 的 Atomic Ptr Plus 项目[1]了解到的技术。清单 7.15 给出了使用这种方法的 push()。

**清单 7.15　在无锁队列中用引用计数 tail 来实现 push()**

```
template<typename T>
class lock_free_queue
{
private:
    struct node;
    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };
    std::atomic<counted_node_ptr> head;
    std::atomic<counted_node_ptr> tail;        ←❶
    struct node_counter
    {
```

[1] Atomic Ptr Plus Project, http://atomic-ptr-plus.sourceforge.net/

```
        unsigned internal_count:30;
        unsigned external_counters:2;          ←❷
    };

    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;       ←❸
        counted_node_ptr next;

        node()
        {
            node_counter new_count;
            new_count.internal_count=0;
            new_count.external_counters=2;     ←❹
            count.store(new_count);

            next.ptr=nullptr;
            next.external_count=0;
        }
    };
public:
    void push(T new_value)
    {
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();
        for(;;)
        {
            increase_external_count(tail,old_tail);     ←❺

            T* old_data=nullptr;
            if(old_tail.ptr->data.compare_exchange_strong(     ←❻
                old_data,new_data.get()))
            {
                old_tail.ptr->next=new_next;
                old_tail=tail.exchange(new_next);
                free_external_counter(old_tail);       ←❼
                new_data.release();
                break;
            }
            old_tail.ptr->release_ref();
        }
    }
};
```

清单 7.15 中，tail 和 head 一样都是 atomic<counted_node_ptr>❶，并且 node 有一个 count 成员代替了之前的 internal_count❸。count 是包含 internal_count 和额外的 external_counters 成员 ❷ 的结构体。注意这里的 external_counters 只包含两个比特，因为最多只有两个计数器。通过使用一个比特来表示它，并且

internal_count 是一个 30 比特的值，总的计数器大小可以保持 32 比特。这就使得在确保整个结构体在 32 比特和 64 比特的机器上都能用一个机器字表示的情况下，还能有足够的范围来表示比较大的内部计数值。为了避免竞争条件，将这些计数作为一个值来更新是很重要的，稍后你将看到。将此结构体保存在一个机器字中在许多平台中使原子操作更容易是无锁的。

node 初始化的时候，internal_count 被设为零，external_counters 的值设为 2❹。因为一旦你将结点添加到队列中，每个新结点都会引用 tail 以及先前结点的next指针。push()与清单 7.14 中类似，除了你为了调用结点data成员的compare_exchange_strong()而解引用从 tail 加载的值之外 ❻，你还调用一个新函数 increase_external_count()来增加计数 ❺，并且之后在旧的 tail 上调用 free_external_counter()❼。

处理好 push()之后，我们来看看 pop()。清单 7.16 展示了它，并且将清单 7.11 中 pop()实现的引用计数逻辑与清单 7.13 中的队列 pop 逻辑结合在一起。

**清单 7.16　从使用引用计数 tail 的无锁队列中将结点出队列**

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        void release_ref();
    };
public:
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed);  <-❶
        for(;;)
        {
            increase_external_count(head,old_head);  <-❷
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                ptr->release_ref();  <-❸
                return std::unique_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next))  <-❹
            {
                T* const res=ptr->data.exchange(nullptr);
                free_external_counter(old_head);  <-❺
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref();  <-❻
        }
    }
};
```

你在开始循环前❶以及增加加载值的外部引用前❷加载 old_head 值。如果 head 与 tail 是同一个结点，那么就可以释放该引用❸并且返回一个空指针，因为队列中没有数据。如果队列中有数据，你就想获得此数据，并且调用 compare_exchange_strong()来实现❹。如同清单 7.11 中的栈一样，它将外部计数和指针作为一个值来比较，如果任何一个改变了，就在释放引用后重新循环❻。如果 compare_exchange_strong()成功了，你就获得了结点中的数据，因此在你释放移出的结点的外部计数后，可以将此值返回给调用者❺。一旦外部引用计数都被释放了并且内部计数值变为零，就可以删除结点了。清单 7.17、清单 7.18 和清单 7.19 展示了处理这些的引用计数函数。

**清单 7.17 释放无锁队列的结点引用**

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        void release_ref()
        {
            node_counter old_counter=
                count.load(std::memory_order_relaxed);
            node_counter new_counter;
            do
            {
                new_counter=old_counter;
                --new_counter.internal_count;        <-❶
            }
            while(!count.compare_exchange_strong(        <-❷
                  old_counter,new_counter,
                  std::memory_order_acquire,std::memory_order_relaxed));

            if(!new_counter.internal_count &&
               !new_counter.external_counters)
            {
                delete this;   <-❸
            }
        }
    };
};
```

node::release_ref()的实现只在清单 7.11 的 lock_free_stack::pop()上改变了一些对应的代码。清单 7.11 中的代码只需要处理一个外部计数，因此可以用一个简单 fetch_sub。而现在即使只想修改 internal_count 域，也需要原子更新整个 count❶。因此需要一个比较/交换循环❷。一旦你减少 internal_count，如果内部计数和外部计数都变为零，那么这就是最后一个引用，就可以安全删除该结点了❸。

**清单 7.18 在无锁队列中获得结点的新引用**

```
template<typename T>
class lock_free_queue
{
private:
    static void increase_external_count(
        std::atomic<counted_node_ptr>& counter,
        counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;
        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!counter.compare_exchange_strong(
              old_counter,new_counter,
              std::memory_order_acquire,std::memory_order_relaxed));
        old_counter.external_count=new_counter.external_count;
    }
};
```

清单 7.18 则相反。这次，你得到一个新的引用并且增加外部计数，而不是释放一个引用。increase_external_count()与清单 7.12 中的 increase_head_count()函数类似，除了它使用一个静态成员函数来将外部计数作为第一个参数来更新，而不是在固定计数器上进行操作。

**清单 7.19 在无锁队列中释放结点的外部计数**

```
template<typename T>
class lock_free_queue
{
private:
    static void free_external_counter(counted_node_ptr &old_node_ptr)
    {
        node* const ptr=old_node_ptr.ptr;
        int const count_increase=old_node_ptr.external_count-2;
        node_counter old_counter=
            ptr->count.load(std::memory_order_relaxed);
        node_counter new_counter;
        do
        {
            new_counter=old_counter;
            --new_counter.external_counters;    <-❶
            new_counter.internal_count+=count_increase;    <-❷
        }
        while(!ptr->count.compare_exchange_strong(    <-❸
              old_counter,new_counter,
              std::memory_order_acquire,std::memory_order_relaxed));
```

```
        if(!new_counter.internal_count &&
           !new_counter.external_counters)
        {
            delete ptr;           <--❹
        }
    }
};
```

与 increase_external_count() 相对的是 free_external_counter()。这与清单 7.11 中 lock_free_stack::pop() 的对应代码是类似的，但是被修改为可以处理 external_counters 计数。它在整个 count 上使用单个 compare_exchange_strong() 来处理两个计数 ❸，正如你在 release_ref() 中减少 internal_count 所做的一样。正如清单 7.11 一样，internal_count 的值被更新了❷，并且 external_counters 的值减少 1❶。如果这两个值现在都为零，那么此结点就没有引用，因此可以安全删除它 ❹。这需要作为一个操作来执行（因此需要比较/交换循环）以避免竞争条件。如果分别更新这两个值，那么两个线程都可能认为它们自己是最后一个线程，因此都删除这个结点，这就会导致未定义的行为。

尽管这种方法是有效的并且是无竞争的，但是它有性能问题。一旦一个线程通过成功完成 old_tail.ptr->data 上的 compare_exchange_strong()（如清单 7.15 中 ❺ 所示），开始执行 push() 操作。此时没有线程可以执行 push() 操作。任何试图执行 push() 操作的线程都会看到新值而不是 nullptr，这就使得 compare_exchange_strong() 失败并且使得线程再次循环。这是一个忙则等待现象，会消耗 CPU 周期而没有任何收益。因此，这是一个锁。第一个调用 push() 的线程阻塞别的线程，直到它完成了操作。因此这个代码不是无锁的。正常情况下，当线程阻塞时，操作系统可以给拥有互斥锁的线程优先权。但是在这里操作系统却无法给拥有互斥锁的线程优先权，因此阻塞的线程会一直消耗 CPU 周期直到第一个线程完成操作。这就需要下一个方法，等待的线程可以帮助正在执行 push() 操作的线程。

### 2. 通过协助另一个线程使得队列无锁

为了使代码无锁，就需要找到一种方法使得即使执行 push() 操作的线程拖延了，等待的线程依然可以继续执行。一种方法就是帮助拖延的线程做它要完成的操作。

在这种情况下，你清楚地知道将要做哪些操作。tail 的 next 指针需要指向一个新的哑元结点，然后更新 tail 指针。哑元结点是没有区别的，因此无论是使用成功将数据入队列的线程创造的哑元结点，还是使用等待将数据入队列的线程创造的哑元结点，都是可以的。如果使得结点的 next 指针成为原子的，就可以使用 compare_exchange_strong() 来设置该指针。一旦设置好 next 指针，就可以在确保它仍然引用同一个最初的结点的情况下，使用 compare_exchange_weak() 循环来设置 tail。如果它没有引用同一个最初的结点，那么就表示别的线程已经更新它了，

此时就停止尝试并且再次循环。这就需要稍微改变 pop()来载入 next 指针。如清单 7.20 所示。

**清单 7.20 修改 pop()来允许帮助 push()**

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;
        std::atomic<counted_node_ptr> next;         <--❶
    };
public:
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_external_count(head,old_head);
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                return std::unique_ptr<T>();
            }
            counted_node_ptr next=ptr->next.load();      <--❷
            if(head.compare_exchange_strong(old_head,next))
            {
                T* const res=ptr->data.exchange(nullptr);
                free_external_counter(old_head);
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref();
        }
    }
};
```

如我所言，这里的改变是很简单的，next 指针现在是原子的 ❶，因此 ❷ 中的 load 也是原子的。在这个例子中，使用了默认的 memory_order_seq_cst 顺序，因此你可以省略明确调用 load()，并且依靠 counted_node_ptr 的隐式载入。但是使用明确的调用可以提醒你稍后在哪里增加明确的内存顺序。

清单 7.21 列出了 push()的更多代码。

**清单 7.21 无锁队列中使用帮助的 push()**

```
template<typename T>
class lock_free_queue
{
```

```
private:
    void set_new_tail(counted_node_ptr &old_tail,              ←❶
                      counted_node_ptr const &new_tail)
    {
        node* const current_tail_ptr=old_tail.ptr;
        while(!tail.compare_exchange_weak(old_tail,new_tail) &&      ←❷
              old_tail.ptr==current_tail_ptr);
        if(old_tail.ptr==current_tail_ptr)                 ←❸
            free_external_counter(old_tail);               ←❹
        else
            current_tail_ptr->release_ref();               ←❺
    }
public:
    void push(T new_value)
    {
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();
        for(;;)
        {
            increase_external_count(tail,old_tail);
            T* old_data=nullptr;
            if(old_tail.ptr->data.compare_exchange_strong(          ←❻
                   old_data,new_data.get()))
            {
                counted_node_ptr old_next={0};
                if(!old_tail.ptr->next.compare_exchange_strong(     ←❼
                       old_next,new_next))
                {
                    delete new_next.ptr;               ←❽
                    new_next=old_next;             ←❾
                }
                set_new_tail(old_tail, new_next);
                new_data.release();
                break;
            }
            else                  ←❿
            {
                counted_node_ptr old_next={0};
                if(old_tail.ptr->next.compare_exchange_strong(      ←⓫
                       old_next,new_next))
                {
                    old_next=new_next;             ←⓬
                    new_next.ptr=new node;         ←⓭
                }
                set_new_tail(old_tail, old_next);          ←⓮
            }
        }
    }
};
```

这与清单 7.15 中的最初的 push() 是类似的，但是也有一些很重要的不同之处。如果你的确设置了 data 指针 ❻，就需要处理这样一种情况。那就是另一个线程已经帮助你了，现在有一个 else 子句也在帮助你❿。

已经设置了结点的 data 指针 ❻，push() 的新版本使用 compare_exchange_strong() 来更新 next 指针❼。使用 compare_exchange_strong() 来避免循环。如果交换失败了，就可以得知另一个线程已经设置了 next 指针，因此就不再需要最初分配的新结点了，就可以删除它❽。你仍然想使用另一个线程更新 tail 设置的 next 值❾。

tail 指针的真正更新发生在 set_new_tail() 中 ❶。这就使用 compare_exchange_weak() 循环❷来更新 tail。因为如果别的线程试图 push() 一个新结点，那么 external_count 的值就发生了改变并且你不想失去它。尽管如此，你要注意如果另一个线程已经成功改变了它，那么你就不能更换此值；否则，就可能以在队列中循环作为结束，而这不是一个好主意。所以，你要确保如果比较/交换失败了，载入值的 ptr 是同样的。如果退出循环时 ptr 是同样的 ❸，那么你就必须成功设置 tail，因此需要释放旧的外部计数 ❹。如果退出循环时 ptr 是不一样的，就说明另一个线程将释放此计数器，因此你只需要通过该线程释放单个引用 ❺。

如果线程调用 push()，并且这次没有成功通过循环设置 data 指针，那么它可以帮助成功的线程完成更新。首先，你尝试更新这个线程新分配结点的 next 指针⓫。如果成功了，你将使用你分配的结点作为新的 tail⓬，并且需要分配另一个新结点预期可以真正入队列⓭。然后你就可以在再次循环前通过调用 set_new_tail 来设置 tail⓮。

你可能已经注意到这一段代码中有很多的 new 和 delete 调用，因为 push() 分配新结点，而 pop() 销毁结点。内存分配器的效率在很大程度上影响了这段代码的性能。一个不好的内存分配器可以完全破坏无锁容器的可扩展性。选择和实现该分配器超出了该书的范围，但是请记住判别分配器是好是坏的唯一办法就是使用它并且测试使用它前后代码的性能。优化内存分配器的通用办法包括在每个线程上都有一个独立的内存分配器，以及使用空闲表来回收结点而不是将它们返回给分配器。

已经举了很多例子了。现在，我们从这些例子里找出写无锁数据结构的一些准则。

## 7.3 编写无锁数据结构的准则

如果你看了本章的所有例子，那么你就会了解使无锁代码正确的复杂性。如果你准备设计你自己的数据结构，那么需要注意一些准则。第 6 章开始部分提到的关于并发数据结构的总体准则依然是适用的，但是你需要更多准则。我将从这些例子中提取出一些有用的准则，当你设计你自己的无锁数据结构时可以参考。

### 7.3.1 准则：使用 std::memory_order_seq_cst 作为原型

std::memory_order_seq_cst 比别的内存顺序更容易理解，因为所有操作形成了总的顺序。在这章的例子中，我们都是从 std::memory_order_seq_cst 开始，并且一旦基础操作正确的情况下才会放松内存顺序约束。从这个意义上来说，使用别的内存顺序是在优化，但是你要避免过早优化。通常，只有当你看到所有代码对数据结构核心操作正确的时候，才能决定哪些操作可以放松。过早地考虑其他内存顺序只会带来麻烦。代码可能会正确工作，但是并不保证是这样的。仅仅跑程序是不够的，除非有个算法检查器来系统测试所有与具体顺序保证相符的可见线程组合。

### 7.3.2 准则：使用无锁内存回收模式

无锁代码最大的问题之一就是管理内存。当别的线程仍然引用对象的时候就不能删除它们，这是最基本的。但是你仍然想尽快删除它们来避免过多的内存消耗。本章将介绍三种方法来确保可以安全回收内存。

- 等待直到没有线程访问该数据结构，并且删除所有等待删除的对象。
- 使用风险指针来确定线程正在访问一个特定的对象。
- 引用计数对象，只有直到没有显著的引用时才删除它们。

在所有的情况下，关键的想法就是使用一些方法来记录有多少线程在访问一个特定的对象，并且只删除不再被引用的对象。有很多方法可以回收无锁数据结构的内存。例如，使用垃圾回收器是很理想的方案。当你不再使用结点的时候，垃圾回收期可以释放结点。在这种情况下写程序就简单一些。

另一个方法就是回收结点，并且当数据结构被销毁的时候才完全释放它们。因为结点是重复使用的，内存永远不会失效。这样避免未定义行为的困难就不存在了。缺点就是另一个问题变得更常见。这就是所谓的 **ABA** 问题。

### 7.3.3 准则：当心 ABA 问题

ABA 问题是任何基于比较/交换的算法都必须提防的问题。它是这样的。

1. 线程 1 读取一个原子变量 x，并且发现它的值为 A。
2. 线程 1 基于这个值执行了一些操作，例如解引用它（如果它是指针的话）或者做一些查找操作。
3. 线程 1 被操作系统阻塞了。
4. 另一个线程在 x 上执行了一些操作，将它的值改为 B。
5. 第三个线程更改了与值 A 相关的值，因此线程 1 持有的数值就不再有效了。这

个变化有可能很大，如释放它所指向的内存或者改变相关的值一样。

6 第三个线程基于新值将 x 的值改回 A。如果这是一个指针，那么就可能是一个新的对象，此对象刚好与先前的对象使用了相同的地址。

7 线程 1 重新取得 x，并在 x 上执行比较/交换操作，与 A 进行比较。比较/交换操作成功了（因为值确实是 A），但是这个 A 的值是错误的。第二步中读取的值不再有效，但是线程 1 并不知道，并且将破坏数据机构。

现在这里没有程序遇到这种问题，但是写无锁程序的时候就很容易遇到这种问题。最常用的避免这种问题的方法就是在变量 x 上使用一个 ABA 计数器。此时，x 加上计数器这样一个结合的数据结构就将作为一个单位，比较/交换就会基于这个单位进行操作。每次修改值的时候，计数器的值都会加一。即使 x 的值是一样的，如果另一个线程修改了 x，比较/交换操作将会失败。

使用空闲表或者回收结点而不是将它返回给分配器，使得 ABA 问题在算法中是很常见的。

### 7.3.4 准则：识别忙于等待的循环以及辅助其他线程

在最后的队列例子中，可以看出执行入队操作的线程必须等待另一个执行入队操作的线程完成操作后才能进行。更不用说，将会出现忙则等待循环，等待的线程不能继续执行的时候会浪费 CPU 时间。如果最终以忙则等待循环结束，那么你事实上就有了阻塞操作，并且也可能会使用互斥元和锁。通过修改程序，如果安排等待中的线程运行的话，那么此线程会在最初的线程完成操作前继续执行未完成的步骤。此时就可以消除忙则等待，并且操作不再被阻塞。在队列的例子中，这就需要将数据成员变为原子变量而不是非原子变量，并且使用比较/交换操作设置它，但是在更复杂的数据结构中需要改变更多。

## 7.4 小结

紧接着第 6 章中描述的基于锁的数据结构，这一章描述了多种使用栈或队列的无锁数据结构的简单实现。你必须注意你的原子操作的内存顺序，确保没有数据竞争并且每个线程看到的数据结构是一致的。你也注意到无锁数据结构中的内存管理比基于锁的数据结构中的内存管理变得更难，并且通过一些方法来处理它。你也注意到如何通过帮助你所等待的线程完成它的操作，从而避免创造等待循环。

设计无锁数据结构是一个很难的任务，并且很容易产生错误，但是在某些情况下，这种数据结构有很好的可扩展性。希望通过本章的例子和准则，你可以设计你自己的无锁数据结构，实现它或者发现别的人写的数据结构中的错误。

如果多个线程共享数据，那么就需要考虑使用什么数据结构以及如何在线程间同步此数据。通过设计并发数据结构，可以将它封装在数据结构中，这样剩下来的代码就可以集中在如何操作此数据结构上而不是数据同步上。在第 8 章中，当我们从并发数据结构转移到并发代码上，你就可以看到它所起的作用了。并行算法使用多线程来提高效率，当算法需要多线程共享数据的时候，选择使用何种并发数据结构就很重要了。

# 第 8 章　设计并发代码

本章主要内容

- 在线程间划分数据的技术
- 影响并发代码性能的因素
- 性能因素如何影响数据结构的设计
- 多线程代码中的异常安全
- 可扩展性
- 几个并行算法实现的示例

前面的章节主要是讨论新的 C++11 工具箱里用来写并行代码的工具。在第 6 章和第 7 章中，我们观察了如何使用这些工具来设计多个线程可以并发存取的安全的基础数据结构。作为木匠，为了制作柜橱或者桌子，不仅仅需要知道如何铰链或者接缝处。同样，需要设计并行代码而不仅仅是设计和使用基础数据结构。你需要了解更广泛的背景，这样就可以构造进行有用工作的更大的结构。我将使用一些 C++标准库算法的多线程实现作为例子，但是同样的原则适用于应用的所有方面。

正如所有编程项目一样，仔细考虑并行代码是很重要的。尽管如此，使用多线程代码比使用顺序代码需要考虑更多的因素。你不仅需要考虑常见的因素，例如封装，耦合和内聚（在很多软件设计书中有详细的描述），而且需要考虑共享哪些数据，如何同步那些数据的存取，哪个线程需要等待哪些别的线程完成特定操作，等等。

本章，我们将致力于这些问题，从高层次考虑使用多少个线程，各个线程执行哪些

代码，以及这些是如何影响代码的透明度。到低层次考虑如何构造共享数据来获得最佳性能。

让我们从在线程间划分工作的技术开始。

## 8.1 在线程间划分工作的技术

设想你被安排建造一所房子。为了完成这项工作，你需要挖地基、筑墙、布线等。理论上说，你可以在足够的训练下全部自己完成，但是将耗费很多时间，并且会一直切换任务。或者，你可以雇佣别人来帮助你。你只需要选择雇佣多少人并且决定他们需要哪些技能。例如，你可以雇佣一些具有一般技能的人，并且让每个人做所有事。你仍然需要切换任务，但是因为人数更多所以能够更快地完成。

或者，你可以雇佣一些专家。例如，砖匠，木匠，电工和管道工。这些专家只做他们专长的事情，因此如果没有管道工程的时候，管道工就不需要做任何事情。事情会比之前完成得更快，因为有更多的人，并且当电工给厨房布线的时候，管道工可以安装卫生间。但是当专家没有工作的时候就会处于等待状态。即使有空闲时间，你也会发现雇佣专家比雇佣一般技能的人工作进展得更快。专家不需要改变工具，并且他们完成任务比一般人要快。无论这种情况是否取决于特殊情况——你都需要尝试一下。

即使你雇佣专家，你仍然可以选择每种专家的数量。例如，雇佣比电工数量更多的砖匠就很合理。如果你要建造不止一座房子的话，你的队伍的构成以及总体效率都会发生改变。即使管道工在给定的房子上不会有太多的工作，你也可以一次建造很多房子，这样他就会始终有工作了。并且，如果当他们没有工作的时候不需要付钱的话，那么就可以负担更大的团队了，即使同一时间只有相同数量的人在工作。

那么线程需要做哪些呢？同样的问题也适用于线程。你需要决定使用多少线程以及它们需要完成什么任务。你需要决定是使用“通用型”线程来在需要的时候执行操作，还是使用“专家型”线程来做好一件事或一些合作的事。你需要决定无论是为了使用并发性而划分的原因，还是如何去做都会对代码的性能和清晰度产生很大影响。因此理解这些选择是很重要的，这样当设计你的应用中的数据结构的时候，就可以做出合适的决定。在这部分，我们将会看到一些划分任务的方法。在我们做别的工作前先来看看如何在线程间划分数据。

### 8.1.1 处理开始前在线程间划分数据

最早并行化的算法是简单的算法，如 `std::for_each` 在数据集合中对每个元素进行操作。为了并行化这样的算法，就需要将每个元素划分到一个处理线程中。如何划分元素来得到最优性能在很大程度上决定于数据结构的细节，稍后我们分析性能问题的

时候就可以看到了。

划分数据最简单的方法就是将第一个 N 元素分配给一个线程，将下一个 N 元素分配给另一个线程，以此类推，正如图 8.1 所示，但是也可以使用别的模式。无论如何划分数据，每个线程只能处理分配给它的元素，并且直到它完成任务的时候才能与别的线程通信。

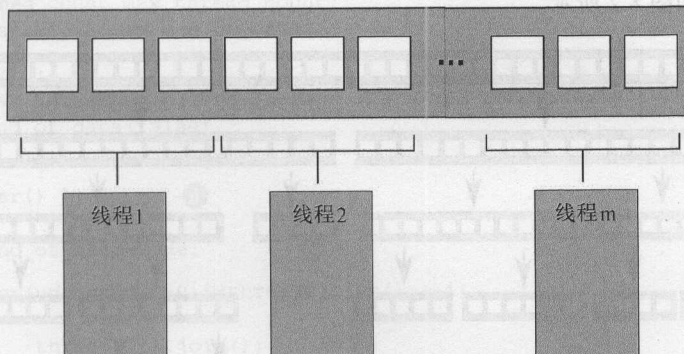



图 8.1 在线程间划分连续数据块

这种结构与使用消息传递接口（Message Passing Interface, MPI）[1]或者 OpenMP[2]框架编程的结构是类似的。一个任务被分成一个并行任务集，工作的线程独立运行这些任务，并且在最后的化简步骤中合并这些结果。这是 2.4 节中 accumulate 例子使用的方法，在这种情况下，并行任务和最后的步骤都是累加。对一个简单的 for_each 来说，最后的步骤是不需要的，因为不需要化简结果。

确定将最后的步骤作为化简步骤是很重要的，清单 2.8 中的简单实现将执行此化简作为最后的线性步骤。尽管如此，这个步骤也可以并行化。累加过程本身就是化简操作，因此可以修改清单 2.8，当线程的数量比线程处理的的最少数据的数量还要多，就可以递归地调用它本身。或者，当每个线程完成它的任务后，工作线程可以执行一些化简操作步骤，而不是每次产一些新线程。

尽管这种方法是很有效的，但是并不适用于所有情况。有时数据不能被事先划分，因为只有当处理数据的时候才知道如何划分。在化简算法例如快速排序中，这种情况更明显。因此需要一个不同的方法。

## 8.1.2 递归地划分数据

快速排序算法有两个基本步骤，基于其中一个元素（关键值）将数据划分为两部分，

---

1 http://www.mpi-forum.org/

2 http://www.openmp.org/

一部分在关键值之前，一部分在关键值之后。然后递归地排序这两部分。你无法通过预先划分数据来实行并行，因为只有当处理元素的时候才知道他们属于哪一个“部分”。如果你打算并行这个算法，就需要把握递归的本质。每次递归的时候，会调用更多的 quick_sort 函数来排序关键点之前和关键点之后的元素。这些递归调用是完全相互独立的，因为它们读取完全不同的元素集合。这种划分可以作为我们初步的候选方案。此递归划分如图 8.2 所示。

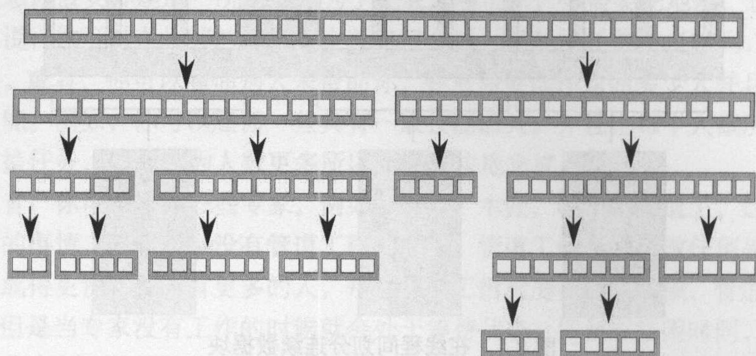


图 8.2 连续划分数据

在第 4 章中，有这样一个实现。不仅只是为这两部分执行两个递归调用，你还在每一步都在前一部分使用 std::async()来生成异步任务。通过使用 std::async()，你让 C++线程库来决定何时在一个新线程上运行这个任务，以及何时同步运行它们。

当你对很大规模的数据进行排序的时候这是很重要的。为每一个递归调用生成一个新线程会很快产生大量线程。当我们考虑性能的时候，你就会发现如果线程太多的时候，就可能降低了应用。如果数据集很大的时候也可能会用完所有的线程。像这样用递归来划分总体任务的方法是很好的，只需要严格控制线程数量就可以了。在简单情况下，std::async( )就可以处理它，但是这并不是唯一选择。

或者使用 std::thread::hardware_concurrency()函数来选择线程数量，正如清单 2.8 中 accumulate()的并行版本所做的一样。然后，你只是将此块存储到第 6 章和第 7 章描述线程安全栈中，而不是为递归调用创建一个新线程。如果线程不在工作，就说明它已经处理完所有块，或者等待存储在栈中的块。此时可以从栈中得到一个块并将它排序。

清单 8.1 列出了使用这种方法的简单实现。

**清单 8.1 使用待排序块栈的并行快速排序**

```
template<typename T>
struct sorter              <—❶
{
```

```
    struct chunk_to_sort
    {
        std::list<T> data;
        std::promise<std::list<T> > promise;
    };
    thread_safe_stack<chunk_to_sort> chunks;        ❷
    std::vector<std::thread> threads;              ❸
    unsigned const max_thread_count;
    std::atomic<bool> end_of_data;
    sorter():
        max_thread_count(std::thread::hardware_concurrency()-1),
        end_of_data(false)
    {}
    ~sorter()    ❹
    {
        end_of_data=true;    ❺
        for(unsigned i=0;i<threads.size();++i)
        {
            threads[i].join();    ❻
        }
    }
    void try_sort_chunk()
    {
        boost::shared_ptr<chunk_to_sort > chunk=chunks.pop();    ❼
        if(chunk)
        {
            sort_chunk(chunk);    ❽
        }
    }
    std::list<T> do_sort(std::list<T>& chunk_data)    ❾
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }

        std::list<T> result;
        result.splice(result.begin(),chunk_data,chunk_data.begin());
        T const& partition_val=*result.begin();

        typename std::list<T>::iterator divide_point=    ❿
            std::partition(chunk_data.begin(),chunk_data.end(),
                [&](T const& val){return val<partition_val;});

        chunk_to_sort new_lower_chunk;
        new_lower_chunk.data.splice(new_lower_chunk.data.end(),
                                     chunk_data,chunk_data.begin(),
                                     divide_point);

        std::future<std::list<T> > new_lower=
```

```
        new_lower_chunk.promise.get_future();
    chunks.push(std::move(new_lower_chunk));    <-⑪
    if(threads.size()<max_thread_count)    <-⑫
    {
        threads.push_back(std::thread(&sorter<T>::sort_thread,this));
    }

    std::list<T> new_higher(do_sort(chunk_data));

    result.splice(result.end(),new_higher);
    while(new_lower.wait_for(std::chrono::seconds(0)) !=
          std::future_status::ready)    <-⑬
    {
        try_sort_chunk();    <-⑭
    }

    result.splice(result.begin(),new_lower.get());
    return result;
  }

  void sort_chunk(boost::shared_ptr<chunk_to_sort > const& chunk)
  {
    chunk->promise.set_value(do_sort(chunk->data));    <-⑮
  }

  void sort_thread()
  {
    while(!end_of_data)    <-⑯
    {
        try_sort_chunk();    <-⑰
        std::this_thread::yield();    <-⑱
    }
  }
};
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)    <-⑲
{
  if(input.empty())
  {
    return input;
  }
  sorter<T> s;

  return s.do_sort(input);    <-⑳
}
```

这里，parallel_quick_sort 函数⑲代表了 sorter 类❶ 的大部分功能，提供了一种简单的方法将未排序块❷ 所在的栈进行分组，以及将线程集❸ 分组。主要的工作是在 do_sort 成员函数❾中完成的，它是用来完成通常的数据排序❿。这次，它将块压入栈中⑪，而不是为一个块产生一个新的线程，并且当你仍然有处理器可以分配的时候就产生一个新进程⑫。因为后一部分可能是被另一个线程处理，你就必须等待它处理完成⑬。为了处理这种情况（即只有一个线程或者别的线程都在工作），你就试图在等待的时候，让这个线程处理栈中的块⑭。try_sort_chunk 只是将一个块出栈❼并且将

它排序❽，将结果存储在 promise 中，此结果可以被将此块放入栈中的线程取得⓯。

当未设置 end_of_data 标志的时候⓱，可以在循环中产生线程将栈中的块先出栈然后排序⓰。在检查的时候，它们让别的线程先对栈进行操作。这段代码依靠 sorter 类析构函数 ❹ 来结束这些线程。当所有数据都被排序了，就会返回 do_sort（即使线程仍然在运行），因此主线程将从 parallel_quici_sort⓴中返回，并且销毁你的 sorter 对象。这就设置了 end_of_data 标志位❺并且等待线程结束 ❻。设置标志位结束了线程函数中的循环⓰。

使用这种方法就不会和使用 spawn_task 来产生一个新线程一样导致无穷多个线程这样的问题了，并且不再和 std::async()一样依赖 C++线程库来选择线程数量。现在我们将线程数量限制到 std::thread::hardware_concurrency()来避免过多的任务切换。尽管如此，你有另一个潜在的问题，处理这些线程和线程间通信给代码增加了很多复杂性。同时，尽管这些线程在处理不相关的数据元素，它们都访问栈来移入和移出所操作的块。即使使用无锁（因此是无阻塞的）栈，这种竞争会降低性能。你稍后会看到原因。

这种方法是一种特殊版本的线程池——有一个线程集，每个线程处理等待列表中的工作，然后回到线程池。线程池存在的问题（包括列表上的竞争），第 9 章中有解决这些问题的方法。本章稍后将讨论如何将程序扩展到多处理器上执行（参见 8.2.1 节）。

在处理开始前划分数据和递归划分数据都是假设数据是固定不变的，然后你寻找划分它的方法，但是情况并不总是这样。如果数据是动态生成的或者是外部输入的，那么这种方法就不行了。在这种情况下，通过任务类型来划分工作比基于数据划分更合适。

### 8.1.3 以任务类型划分工作

通过给每个线程分配不同数据块在线程间划分工作（无论是事先划分还是处理过程中递归划分）仍然是基于这样的假设，即线程将会基于每个数据块做同样的工作。划分工作的另一种方法是使得线程变得专业化，即每个线程执行不同的任务，就如同建造房子的时候管道工和电工执行不同的任务一样。线程可以基于也可以不基于同样的数据来工作，但是如果基于同样的数据，那也是有不同的目的。

这种划分工作的方式源自于将并发中的关注点分离。每个线程都有不同的任务，并且独立于别的线程来工作。偶尔别的线程可能给它数据或者有触发事件需要处理，但是通常每个线程只做一件事情。这本质上是一个好设计，每块任务都应该只负责一个单一的任务。

#### 1. 以任务类型划分工作来分离关注点

当在一段时间内需要持续运行多个任务的时候，或者需要此应用能够及时处理

有输入的事件（例如用户键盘输入或者输入网络数据）而不影响别的线程继续执行的时候，单线程应用需要处理与单一任务原则间的矛盾。在单线程环境中，你手工写代码来执行任务A的一部分，执行任务B的一部分，检查键盘输入，检查输入的网络包，然后继续循环执行A的另一部分。这就意味着任务A代码的结束部分会变复杂，因为需要保留它的状态以及周期性地返回控制给主循环。如果你给循环增加了太多任务，运行就会变得很慢，并且使用者会发现键盘输入的响应时间太长。你肯定见过一些应用采取过极端的方式。你设置它处理一些任务，然后接口保持不变直到它完成任务。

这就是线程发生作用的地方。如果你在独立的线程里运行每个任务，操作系统就可以帮你处理这种问题。在任务A的代码中，你可以致力于执行任务，并且不需要担心保留状态以及返回到主循环或者在此之前你花费了多长时间。操作系统将自动地保留状态，然后在适当的时候切换到任务B或C，并且如果系统有多个核或者处理器，任务A和B就可以真正地并行运行。处理键盘输入或者网络包的代码将会运行得很及时，并且皆大欢喜。使用者获得及时响应，你作为开发者可以写更简单的代码，因为每个线程都致力于做与任务直接相关的操作，而不是与控制流和用户互动混合在一起。

看上去这是一个很好的版本。它真的如此吗？如同任何事情一样，它取决于细节。如果每件事情都是独立的，并且线程不需要与另一个线程通信，那么它就确实很简单了。可是，事实却并不是如此。这些后台任务经常做一些用户需要的事情，并且它们需要更新用户接口让用户知道是何时完成这些任务的。或者，用户可能要取消任务，这就会要求用户接口以某种方式给后台任务发送一个消息通知它停止此任务。这两种情况都需要仔细的思考，设计以及适当的同步。但是这些关注点都是分离的。用户接口线程仍然只是处理用户接口，但是当别的线程要求时，它需要更新它的接口。同样，运行后台任务的线程仍然致力于那个任务要求的操作，只有当它的操作是“允许另一个线程停止此任务”。在这两个例子中，线程都不关心该要求是从哪里产生的，只关心它是否是为它们准备的以及与它们的任务是否直接相关。

多个线程关键点分离有两个危害。首先就是你将分离错误的关键点。征兆就是线程间有很多共享数据，或者不同的线程都以等待彼此作为结束。这两种情况都可以归结为线程间有太多的通信。如果发生这种情况，就值得查找产生通信原因。如果所有的通信都与同一件事相关，那么可能那就是单线程的关键任务，并且从所有引用它的线程中获得。或者，如果两个线程彼此之间需要很多通信但是与别的线程通信很少，那么它们就应该联合为一个线程。

当根据任务类型在线程间划分工作的时候，你就不需要局限于完全独立的任务。如果多个数据集合需要应用同样的操作序列，那么就可以划分工作使每个线程执行整个序列中一个步骤。

2. 划分线程间的任务序列

如果你的任务是由在很多独立数据项上运行同样的操作序列组成的话，就可以使用管道来开发系统可能的并发性。可以将它类比为管道，数据通过一系列操作（管子）从一端流入，并且从另一端流出。

为了用这种方式划分工作，你在管道的每一个步骤都创造一个独立的线程——序列中的每个操作都有一个线程。当操作完成时，数据元素被放入队列中供下一个线程获得。这就允许当管道中第二个线程在操作第一个元素的时候，第一个线程执行序列中的第一个操作来开始下一个数据元素。

这是仅仅在线程间划分数据的一种替代方法，正如 8.1.1 节中描述的一样，并且在操作开始时并不知道所有输入数据的情况下是适用的。例如，数据可能是通过网络输入的，或者序列的第一个操作就是扫描一个文件系统来识别要处理的文件。

当序列中的每个操作都消耗时间的时候，管道也可以很好地工作。通过在线程间划分任务而不是数据，你改变了性能概况。假设你要在四核上处理 20 个数据项，并且每个数据项需要四个步骤，每个步骤需要 3 秒。如果你在四个线程中划分数据，那么每个线程要处理 5 个数据项。假设没有别的影响时间的处理，12 秒后将处理完 4 个数据项，24 秒后将处理完 8 个数据项，以此类推。1 分钟后将处理完所有的 20 个数据项。如果使用管道，事情就会不一样了。可以将四个步骤中的每个步骤分配到一个处理核上。现在每个核心都要处理第一个元素，因此需要 12 秒。实际上，12 秒后你只处理完一个数据项，这就没有比数据划分的方法好。但是，一旦管道被使用，处理事情就会变得不一样了。在第一个核心处理完的第一项以后，它接着处理第二项。因此一旦最后一个核处理完第一项，它就可以在第二项上执行它的步骤。现在每 3 秒都可以处理完一个数据项，而不是在每 12 秒能处理完一批四个数据项。

处理整个分批会花费更长的时间，因为在最后一个核开始处理第一个数据项之前你需要等待 9 秒。但是更平滑，更有规律的处理在某些环境下可能会很有效。例如，考虑用来收看高清数字电视的系统。为了使电视是可看的，你至少需要每秒 25 帧并且更多的帧得到更理想的效果。同样，观看者需要它们被均匀地分开来得到持续活动的印象；一个可以每秒解码 100 帧的应用是没用的，如果它暂停一秒，然后显示 100 帧，然后再停一秒，然后显示另一个 100 帧；另一方面，观看者可能很高兴接受当他们开始观看电视的时候有几秒的延迟。在这种情况下，使用管道以一个更好、更稳定的速度并行输出帧可能会更好。

已经看过在线程间划分工作的一些方法，我们来看看影响多线程系统性能的因素以及它是如何影响你选择的方法的。

## 8.2 影响并发代码性能的因素

如果要用并发来提高程序在多处理器环境下的性能，我们需要了解哪些因素会影响。哪怕你只是用多线程来进行关注点分离，你需要确保这不会对性能有负面影响。如果你的程序在 16 核的机器上跑得比在一台老的单核机器上还更慢，客户可是不会买账的。

接下来，我们会看到有非常多的因素影响多线程程序的性能——哪怕只是改变下每个线程处理的哪部分数据（其他都保持不变）都会对性能有巨大的影响。我们先不进一步展开，从看其中一些明显的因素看起，如你的目标机器有多少个处理器？

### 8.2.1 有多少个处理器？

处理器的数量和结构是多线程程序的性能的首要和关键的因素。有时你在开发时是知道目标硬件，有目标硬件的规格甚至在一样的硬件上开发。有这种条件你算得上是幸运儿了，但是一般情况我们没这种待遇。也许你是在相似的硬件环境下开发，但是其中的差异会很致命。例如，你在 2 核或 4 核的系统下开发，而你的客户可能有多核处理器或者多个单核处理器，乃至多个多核处理器。并发程序的行为和性能在这样不同的环境下会有很大的差异。因此你需要仔细考量会有哪些影响并尽可能地进行测试。

简单近似的话，一个 16 核处理器等价于 4 个 4 核处理器或 16 个单核处理器，因为它们都可以并发执行 16 个线程。你的程序至少要有 16 个线程来利用好这些硬件。如果少于 16，就会有处理器性能闲置（除非这个机器还在运行其他程序，我们现在忽略这个情形）；另一方面，如果你有多于 16 个线程要运行（没有阻塞或等待），会浪费处理器的运算力在切换这些线程上（参见第 1 章）。这种情况一般被称为过度订阅（**oversubscription**）。

为了让程序中的线程数量随着硬件能同时运行的线程数量扩展，C++11 的标准库提供了 `std::thread::hardware_concurrency()`。我们已经看过使用它的例子。

直接调用 `std::thread::hardware_concurrency()`时需要注意，你的程序并没有考虑机器上运行的其他线程，除非你显式地共享这些信息。最坏的情况是，多个线程同时调用 `std::thread::hardware_concurrency()`会造成严重的过度订阅。而 `std::async()`会在被调用时，由标准库处理所有这些调用并适当地调度，从而避免这个问题。精心设计的线程池也能避免这个问题。

即使你已经考虑了程序中所有运行的线程，你仍然会被其他同时运行的程序影响。尽管在单用户环境下很少有多个 CPU 密集型任务同时运行，在某些场景下这种情况会更普遍。为这种应用场景设计的系统一般会提供某种机制来让程序选择合适的线程数，

当然这已经不在 C++标准内了。一种方法是提供类似 std::async()的调用，在选择线程数量时考量所有程序异步执行的任务数量。另一种是限制给定程序使用的核的数量。我希望在这样的平台上可以用 std::thread::hardware_concurrency()来返回这个数量，不过这取决于具体的系统。如果你需要处理这样的情形，可以去查阅文档了解目标系统提供了哪种方案。

这种情况下随之而来的麻烦是：一个问题的理想算法取决于问题大小和处理单元的数量。如果你在有大量处理单元的大规模并行处理机上运行，耗费操作多的算法可能会比操作少的算法快得多，因为每个处理器只需要处理少量的操作。

随着处理器数量增加，另一个影响性能的问题也出现了，多个处理器访问相同的数据。

## 8.2.2 数据竞争和乒乓缓存

如果两个线程同时在不同的处理器上运行，它们同时读取同样的数据通常不会有问题，数据会被复制到各自的缓存，两个处理器都可以继续执行。但是，如果其中一个线程修改了数据，这个修改需要花费时间传播到另一个处理器的缓存。取决于两个线程的操作和这些操作的内存顺序，这样的修改可能导致第二个处理器停下来等待内存硬件传播对数据的修改。从 CPU 指令来说，这是个相当于数百条指令的显著缓慢的操作，具体的时间主要与硬件的物理结构相关。

考虑下面这段简单代码。

```
std::atomic<unsigned long> counter(0);
void processing_loop()
{
    while(counter.fetch_add(1,std::memory_order_relaxed)<100000000)
    {
        do_something();
    }
}
```

这里的 counter 是全局的，每个调用 processing_loop()的线程都在修改同一个变量。因此，每次在增加时，处理器必须保证它的缓存中有 counter 的最新拷贝、修改，然后发布到其他处理器。即使你用 std::memory_order_relaxed 来让编译器不同步其他数据，fetch_add 是一个“读-修改-写”操作，因此需要获取变量最新的值。如果其他处理器上的其他线程在运行同样的代码，counter 的数据就必须在两个处理器之间来回传递来保证每个处理器在增加时都有最新的 counter 值。如果 do_something()耗时很少，或者有太多的处理器在运行这段代码，处理器可能会处于互相等待的状态。一个处理器已经准备好更新这个值，但是另一个处理器已经在做了，这就要等待另一个处理器更新，并且这个改动已经传播完成，这种情况被称为高竞争

（**high contention**）。如果处理器很少需要互相等待，则称为**低竞争**（**low contention**）。

在这样的循环中，counter 的数据在各处理器的缓存间来回传递。这被称为**乒乓缓存**（**cache ping-pong**），而且会严重影响程序的性能。如果处理器因为需要等待缓存而被挂起，在这个时间里处理器无法进行任何工作，即使有其他线程等待被执行，这对整个程序来说不是个好消息。

也许你会觉得这不会在自己身上发生，因为不会写这样的循环。但是你能确定吗？如互斥锁，如果你在一个循环中获得一个互斥元，你的代码从数据访问的角度看和上面的代码会非常相像。为了锁住互斥元，另一个线程必须从它所在的处理器获得互斥元并修改。当操作完成后，它又修改互斥元来释放，相关的数据必须传递到下一个需要互斥元的线程所在的处理器。这个传递所需的时间是第二个线程等待第一个线程释放互斥元的额外时间。

```
std::mutex m;
my_data data;
void processing_loop_with_mutex()
{
    while(true)
    {
        std::lock_guard<std::mutex> lk(m);
        if(done_processing(data)) break;
    }
}
```

现在是最棘手的部分：如果数据和互斥元被不止一个线程访问，当你添加更多的核和处理器时，你就越可能面临高竞争，处理器需要等待另一个处理器。如果你更快地用多线程来处理同样的数据，这些线程会竞争这些数据，并竞争同一个互斥元。线程数量越多，就越可能同时试图获取互斥元或者访问某个原子变量。

竞争互斥元的影响通常和竞争原子操作不同，因为使用互斥元在操作系统层面将线程串行化，而不是在处理器层面。如果你有足够的线程等待运行，操作系统会在一个线程等待互斥元时调度另一个线程运行。与之相对的是，处理器的挂起会阻止其他线程在这个处理器上运行。但是，这仍然会影响其他竞争这个互斥元的线程的性能，因为它们每次只有一个会被运行。

在第 3 章，我们看过如何用一个单写入者，多读取者的互斥元保护很少更新的数据结构的例子（参见 3.3.2 节）。乒乓缓存会使得只用一个互斥元的好处不明显，特别是工作量大的时候。因为所有访问数据的线程（甚至是读取者仍然需要自己去修改互斥元。随着访问数据的处理器数量上升，互斥元本身的竞争也在增加，包含互斥元的缓存线必须在各个核之间传递，导致获取和释放锁的时间不可接受。你可以用一些方法来改善，主要是通过将互斥元分布在多个缓存线，但这就意味着你要自己去实现这样的互斥元，而不能使用系统本身提供的。

如果乒乓缓存效应有害，我们如何避免呢？本章稍后会揭示，解决方法依赖于提高

并发度，尽可能地避免两个线程竞争从一个内存位置。不过这并不容易做到，即使一个特定内存区域只有一个线程会去访问，你仍然会遇到乒乓缓存，因为存在**假共享**（**false sharing**）的问题。

### 8.2.3 假共享

处理器缓存的最小单位通常不是一个内存地址，而是一小块称为**缓存线**（**cache line**）的内存。这些内存块一般大小为 32～64 字节，取决于具体的处理器。缓存只能处理缓存线大小的内存块，相邻地址的数据会被载入同一个缓存线。有时这是好事，线程访问的数据在同一个缓存线比分布在多个缓存线更好。但是如果缓存线内有不相关但需要被别的线程访问的数据，会导致严重的性能问题。

假设你有一个 `int` 型的数组以及一组线程，每个线程都不停访问和改写数组中彼此正交的部分。因为整型的大小通常小于缓存线，数组中的多个元素会出现在同一个缓存线。这样即使线程只访问自己相关的数据，仍然会有乒乓缓存。一个线程在更改其访问的数据时，缓存线的所有权需要转移到其所在的处理器，而另一个线程所需的数据可能也在这个缓存线上，当它访问时缓存线又要再次转移。这个缓存线是两者共享的，然而其中的数据并不共享，因此被称为**假共享**（**false sharing**）。这里的解决方案是构造好数据的结构，使得被同一个线程访问的数据在内存中也是相邻的，这样就更可能出现在同一个缓存线，而不同线程访问的数据则分散在内存中，使之更可能地出现在不同的缓存线。本章稍后会介绍如何根据这个要求设计数据和代码。

如果说多个线程访问同一个缓存线有害，那么单个线程访问的数据的内存布局又有什么影响呢？

### 8.2.4 数据应该多紧密

假共享是由于一个线程访问的数据与另一个线程的靠得太近，而另一个与数据布局直接相关的性能隐患则来自一个线程本身。根源是数据的相邻度。如果线程访问的数据分散在内存中，意味着这些数据分布在各个缓存线上。因此，更多的缓存线需要加载到处理器的缓存中，这会增加内存访问延迟，性能要低于数据分布紧密的情况。

同时，这也会增加线程需要的某个缓存线同时含有其他线程访问的数据的可能性。极端情况下，缓存中无关的数据会多于你关心的数据。这会浪费宝贵的缓存空间，迫使处理器将需要的数据移出缓存来腾出空间，这样更容易缓存未命中而不得不从内存中获取数据。

这对单线程代码的性能很重要，而我们在这里考虑它的原因是**任务切换**（**task switching**）。如果有多余 CPU 核数量的线程，每个核都将运行多个线程。这会增加缓存

的压力，因为你要保证不同线程访问不同的缓存线以避免假共享。因此，当处理器切换线程时，数据分散在多个缓存线比每个线程的数据都紧靠在同一个缓存线，更可能需要重载这些缓存线。

如果线程数多于核或者处理器处理，操作系统可能也会选择在一个核上给某个线程分配一个时间片，之后又到另一个核上给这个线程分配时间片。这就需要将这个线程所需的缓存线从第一个核转移到第二个核。需要转移的缓存线越多，消耗的时间也越多。尽管操作系统通常会尽可能避免这种情况，这种现象仍然存在并且一旦发生就会严重影响性能。

任务切换导致的问题在大量线程处于**就绪**而不是**等待**状态时特别突出。这是我们已经接触过的问题：过度订阅。

### 8.2.5 过度订阅和过多的任务切换

在多线程的系统中，线程数量通常会多于处理器数量，除非你使用的是**大规模并行**处理机。然而，线程经常花时间等待外部 I/O 操作完成或者因为互斥元而阻塞，又或者在等待一个条件变量等，因此数量多于处理器并不会带来问题。多出的线程可以让程序进行有用的工作而不是使处理器空闲等待。

但是这不总是好事。当你有**太多**的线程时，你会有多余可用处理器的**就绪**线程，操作系统将会开始频繁的任务切换以保证所有线程享有适当的时间片。我们在第 1 章看到过，这会增加任务切换的额外开销，并且由于数据没有相邻导致的一系列缓存问题。过度订阅会在以下情况产生：你有任务无限制地生成新的线程，如第 4 章递归调用的快速排序；或者你根据任务类型分配的线程数量大于处理器的数量，而任务更依赖于 CPU 而不是 I/O。

如果你只是因为划分数据产生了太多的线程，你可以简单的限制工作线程的数量，就像我们在 8.1.2 节见过的一样。如果过度订阅来自于对任务类型的划分，你就没有什么改进的余地了，这时选择合适的划分也许超出了你对目标平台的知识储备，除非性能无法接受而且能证明对划分的改变确实可以提高性能才值得去做。

其他因素也能影响多线程代码的性能。乒乓缓存的代价在两个单核处理器和一个双核处理器上会有很大的差异，哪怕两个平台的 CPU 类型和时钟频率都一样。以上都是重要的因素，对性能有显著的影响。现在，让我们了解一下这会如何影响我们代码和数据结构的设计。

## 8.3 为多线程性能设计数据结构

在 8.1 节中我们看到了在线程间划分工作的一些方法，在 8.2 节中我们看到了影响

代码性能的一些因素。当设计多线程性能的数据结构的时候如何使用这些信息呢？这是在第 6 章和第 7 章中处理的很困难的问题，是关于设计可以安全并行读取的数据结构。正如你在 8.2 节中看到的一样，即使没有别的线程共享此数据，单个线程使用的数据布局也会对它产生影响。

当为多线程性能设计你的数据结构时需要考虑的关键问题是竞争、假共享以及数据接近。这三个方面都会对性能产生很大影响，并且通常你可以通过改变数据布局或者改变分配给某线程的数据元素来提高性能。首先，我们来看一个简单的例子，在线程间划分数组元素。

## 8.3.1 为复杂操作划分数组元素

假设你正在做一些复杂的数学计算，你需要将两个大矩阵想乘。为了实现矩阵相乘，你将第一个矩阵的第一行每个元素与第二个矩阵的第一列相对应的每个元素相乘，并将结果相加得到结果矩阵左上角第一个元素。然后你继续将第二行与第一列相乘得到结果矩阵第一列的第二个元素，以此类推。正如图 8.3 所示，突出显示的部分表明了第一个矩阵的第二行与第二个矩阵的第三列配对，得到结果矩阵的第三列第二行的值。

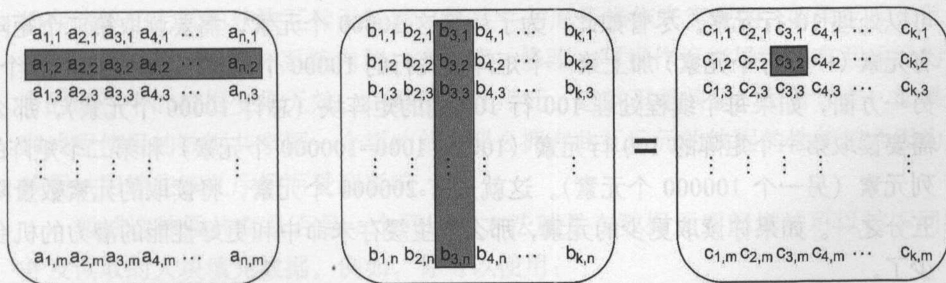


图 8.3 矩阵相乘

为了值得使用多线程来优化该乘法运算，现在我们假设这些都有几千行和几千列的大矩阵。通常，非稀疏矩阵在内存中是用一个大数组表示的，第一行的所有元素后面是第二行的所有元素，以此类推。为了实现矩阵相乘，现在就有三个大数组了。为了获得更优的性能，你就需要注意数据存取部分，特别是第三个数组。

有很多在线程间划分工作的方法。假设你有比处理器更多的行/列，那么你就可以让每个线程计算结果矩阵中某些列的值，或者让每个线程计算结果矩阵中某些行的值，或者甚至让每个线程计算结果矩阵中规则矩形子集的值。

回顾 8.2.3 节和 8.2.4 节，你就会发现读取数组中的相邻元素比到处读取数组中的值要好，因为这样减少了缓存使用以及假共享。如果你使每个线程处理一些列，那么就需要读取第一个矩阵中的所有元素以及第二个矩阵中相对应的列中元素，但是你只会得到

列元素的值。假设矩阵是用行顺序存储的，这就意味着你从第一行中读取 $N$ 个元素，从第二行中读取 $N$ 个元素，以此类推（$N$ 的值是你处理的列的数目）。别的线程会读取每一行中别的元素，这就很清楚你应该读取相邻的的列，因此每行的 $N$ 个元素就是相邻的，并且最小化了假共享。当然，如果这 $N$ 个元素使用的空间与缓存线的数量相等的话，就不会有假共享，因为每个线程都会工作在独立的缓存线上。

另一方面，如果每个线程处理一些行元素，那么就需要读取第二个矩阵中的所有元素，以及第一个矩阵中相关的行元素，但是它只会得到行元素。因为矩阵是用行顺序存储的，因此你现在读取从 $N$ 行开始的所有元素。如果你选择相邻的行，那么就意味着此线程是现在唯一对这 $N$ 行写入的线程；它拥有内存中连续的块，并且不会被别的线程访问。这就比让每个线程处理一些列元素更好，因为唯一可能产生假共享的地方就是一块的最后一些元素与下一个块的开始一些元素。但是值得花时间确认目标结构。

第三种选择—划分为矩形块如何呢？这可以被看做是先划分为列，然后划分为行。它与根据列元素划分一样存在假共享问题。如果你可以选择块的列数目来避免这种问题，那么从读这方面来说，划分为矩形块有这样的优点：你不需要读取任何一个完整的源矩阵。你只需要读取相关的目标矩阵的行与列的值。从具体方面来看，考虑两个 1000 行和 1000 列的矩阵相乘。就有一百万个元素。如果你有 100 个处理器，那么每个线程可以处理 10 行元素。尽管如此，为了计算这 10000 个元素，需要读取第二个矩阵的所有元素（一百万个元素）加上第一个矩阵相关行的 10000 个元素，总计 1010000 个元素；另一方面，如果每个线程处理 100 行 100 列的矩阵块（总计 10000 个元素），那么它们需要读取第一个矩阵的 100 行元素（100 x 1000=100000 个元素）和第二个矩阵的 100 列元素（另一个 100000 个元素）。这就只有 200000 个元素，将读取的元素数量降低到五分之一。如果你读取更少的元素，那么发生缓存未命中和更好性能的潜力的机会就更少了。

因此将结果矩阵划分为小的方块或者类似方块的矩阵比每个线程完全处理好几行更好。当然，你可以调整运行时每个块的大小，取决于矩阵的大小以及处理器的数量。如果性能很重要，基于目标结构分析各种选择是很重要的。

你也有可能不进行矩阵乘法，那么它是否适用呢？当你在线程间划分大块数据的时候，同样的原则也适用于这种情况。仔细观察数据读取方式，并且识别影响性能的潜在原因。在你遇到的问题也可能有相似的环境，就是只要改变工作划分方式可以提高性能而不需要改变基本算法。

好了，我们已经看到数组读取方式是如何影响性能的。其他数据结构类型呢？

## 8.3.2 其他数据结构中的数据访问方式

从根本上说，当试图优化别的数据结构的数据访问模式时也是适用的。

- 在线程间改变数据分配，使得相邻的数据被同一个线程适用。
- 最小化任何给定线程需要的数据。
- 确保独立的线程访问的数据相隔足够远来避免假共享。

当然，运用到别的数据结构上是不容易的。例如，二叉树本来就很难用任何方式来再分，有用还是没用，取决于树是如何平衡的以及你需要将它划分为多少个部分。同样，树的本质意味着结点是动态分配的，并且最后在堆上不同地方。

现在，使数据最后在堆上不同地方本身不是一个特别的问题，但是这意味着处理器需要在缓存中保持更多东西。实际上这可以很有利。如果多个线程需要遍历树，那么它们都需要读取树的结点，但是如果树的结点至包含指向该结点持有数据的指针，那么当需要的时候，处理器就必须从内存中载入数据。如果线程正在修改需要的数据，这就可以避免结点数据与提供树结构的数据间的假共享带来的性能损失。

使用互斥元保护数据的时候也有同样的问题。假设你有一个简单的类，它包含一些数据项和一个互斥元来保护多线程读取。如果互斥元和数据项在内存中离得很近，对于使用此互斥元的线程来说就很好，它需要的数据已经在处理器缓存中了，因为为了修改互斥元已经将它载入了。但是它也有一个缺点：当第一个线程持有互斥元的时候，如果别的线程试图锁住互斥元，它们就需要读取内存。互斥元的锁通常作为一个在互斥元内的存储单元上试图获取互斥元的读—修改—写原子操作来实现的，如果互斥元已经被锁的话，就接着调用操作系统内核。这个读—修改—写操作可能导致拥有互斥元的线程持有的缓存中的数据变得无效。只要使用互斥元，这就不是问题。尽管如此，如果互斥元和线程使用的数据共享同一个缓冲线，那么拥有此互斥元的线程的性能就会因为另一个线程试图锁住该互斥元而受到影响。

测试这种假共享是否是一个问题的方法就是在数据元素间增加可以被不同的线程并发读取的大块填充数据。例如，你可以使用：

```
struct protected_data
{
    std::mutex m;
    char padding[65536];
    my_data data_to_protect;
};
```

65536 字节是为了显著大于缓存线

来测试互斥元竞争问题或者使用：

```
struct my_data
{
    data_item1 d1;
    data_item2 d2;
    char padding[65536];
};
my_data some_array[256];
```

来测试数组数据是否假共享。如果这样做提高了性能，就可以得知假共享确实是一

个问题，并且你可以保留填充数据或者通过重新安排数据读取的方式来消除假共享。

当然，当设计并发性的时候，不仅需要考虑数据读取模式，因此让我们来看看别的需要考虑的方面。

## 8.4 为并发设计时的额外考虑

本章我们看了一些在线程间划分工作的方法，影响性能的因素，以及这些因素是如何影响你选择哪种数据读取模式和数据结构的。但是，设计并发代码需要考虑更多。你需要考虑的事情例如异常安全以及可扩展性。如果当系统中处理核心增加时性能（无论是从减少执行时间还是从增加吞吐量方面来说）也增加的话，那么代码就是可扩展的。从理论上说，性能增加是线性的。因此一个有 100 个处理器的系统的性能比只有一个处理器的系统好 100 倍。

即使代码不是可扩展的，它也可以工作。例如，单线程应用不是可扩展的，异常安全是与正确性有关的。如果你的代码不是异常安全的，就可能会以破碎的不变量或者竞争条件结束，或者你的应用可能因为一个操作抛出异常而突然终止。考虑到这些，我们将首先考虑异常安全。

### 8.4.1 并行算法中的异常安全

异常安全是好的 C++代码的一个基本方面，使用并发性的代码也不例外。实际上，并行算法通常比普通线性算法需要你考虑更多关于异常方面的问题。如果线性算法中的操作抛出异常，该算法只要考虑确保它能够处理好以避免资源泄漏和破碎的不变量。它可以允许扩大异常给调用者来处理。相比之下，在并行算法中，很多操作在不同的线程上运行。在这种情况下，就不允许扩大异常了，因为它在错误的调用栈中。如果一个函数大量产生以异常结束的新线程，那么该应用就会被终止。

作为一个具体的例子，我们来回顾清单 2.8 中的 parallel_accumulate 函数，清单 8.2 中会做一些修改。

**清单 8.2 std::accumulate 的并行版本（来自清单 2.8）**

```
template<typename Iterator,typename T>
struct accumulate_block
{
    void operator()(Iterator first,Iterator last,T& result)
    {
        result=std::accumulate(first,last,result);          <-❶
    }
};

template<typename Iterator,typename T>
```

```
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);   ⟵❷

    if(!length)
        return init;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;

    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();

    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);

    unsigned long const block_size=length/num_threads;

    std::vector<T> results(num_threads);   ⟵❸
    std::vector<std::thread>  threads(num_threads-1);   ⟵❹

    Iterator block_start=first;   ⟵❺
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;   ⟵❻
        std::advance(block_end,block_size);
        threads[i]=std::thread(   ⟵❼
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
        block_start=block_end;   ⟵❽
    }
    accumulate_block()(block_start,last,results[num_threads-1]);   ⟵❾

    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));

    return std::accumulate(results.begin(),results.end(),init);   ⟵❿
}
```

现在我们检查并且确定抛出异常的位置：总的说来，任何调用函数的地方或者在用户定义的类型上执行操作的地方都可能抛出异常。

首先，你调用 distance❷，它在用户定义的迭代器类型上执行操作。因为你还没有进行任何工作，并且这是在调用线程上，所以这是没问题的。下一步，你分配了 results 迭代器 ❸ 和 threads 迭代器 ❹。同样，这是在调用线程上，并且你没有做任何工作或者生产任何线程，因此这是没问题的。当然，如果 threads 构造函数抛出异常，那么就必须清楚为 results 分配的内存，析构函数将为你完成它。

跳过 block_start 的初始化 ❺ 因为这是安全的，就到了产生线程的循环中的操作 ❻、❼、❽。一旦在❼中创造了第一个线程，如果抛出异常的话就会很麻烦，你的新 std::thread 对象的析构函数会调用 std::terminate 然后中止程序。

调用 accumulate_block❾也可能会抛出异常，你的线程对象将被销毁并且调用

std::terminate；另一方面，最后调用 std::accumulate❿的时候也可能抛出异常并且不导致任何困难，因为所有线程将在此处汇合。

这不是对于主线程来说的，但是也可能抛出异常，在新线程上调用 accumulate_block 可能抛出异常❶。这里没有任何 catch 块，因此该异常将被稍后处理并且导致库调用 std::terminate()来中止程序。

即使不是显而易见的，这段代码也不是异常安全的。

### 1. 增加异常安全性

好了，我们识别出了所有可能抛出异常的地方以及异常所造成的不好影响。那么如何处理它呢？我们先来解决在新线程上抛出异常的问题。

在第 4 章中介绍了完成此工作的工具。如果你仔细考虑在新线程中想获得什么，那么很明显当允许代码抛出异常的时候，你试图计算结果来返回。std::packaged_task 和 std::future 的组合设计是恰好的。如果你重新设计代码来使用 std::packaged_task，就以清单 8.3 中的代码结束。

**清单 8.3　使用 std::packaged_task 的 std::accumulate 的并行版本**

```
template<typename Iterator,typename T>
struct accumulate_block
{
    T operator()(Iterator first,Iterator last)          <-❶
    {
        return std::accumulate(first,last,T());          <-❷
    }
};
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return init;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;

    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();

    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);

    unsigned long const block_size=length/num_threads;

    std::vector<std::future<T> > futures(num_threads-1);       <-❸
    std::vector<std::thread> threads(num_threads-1);
```

```
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        std::packaged_task<T(Iterator,Iterator)> task(    <--❹
            accumulate_block<Iterator,T>());               ❺
        futures[i]=task.get_future();                     <-┘
        threads[i]=std::thread(std::move(task),block_start,block_end);  <--❻
        block_start=block_end;
    }
    T last_result=accumulate_block()(block_start,last);   <--❼

    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));

    T result=init;                                         <--❽
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get();                          <--❾
    }
    result += last_result;                                 <--❿
    return result;
}
```

第一个改变就是，函数调用 accumulate_block 操作直接返回结果，而不是返回存储地址的引用 ❶。你使用 std::packaged_task 和 std::future 来保证异常安全，因此你也可以使用它来转移结果。这就需要你调用 std::accumulate❷ 明确使用默认构造函数 T 而不是重新使用提供的 result 值，不过这只是一个小小的改变。

下一个改变就是你用 futures 向量 ❸，而不是用结果为每个生成的线程存储一个 std::future<T>。在生成线程的循环中，你首先为 accumulate_block 创造一个任务 ❹。std::packaged_task<T(Iterator,Iterator)>声明了有两个 Iterator 并且返回一个 T 的任务，这就是你的函数所做的。然后你将得到任务的 future❺，并且在一个新的线程上运行这个任务，输入要处理的块的起点和终点 ❻。当运行任务的时候，将在 future 中捕捉结果，也会捕捉任何抛出的异常。

既然你已经使用了 future，就不再有结果数组了，因此必须将最后一块的结果存储在一个变量中❼而不是存储在数组的一个位置中。同样，因为你将从 future 中得到值，使用基本的 for 循环比使用 std::accumulate 要简单，以提供的初始值开始❽，并且将每个 future 的结果累加起来❾。如果相应的任务抛出异常，就会在 future 中捕捉到并且调用 get()时会再次抛出异常。最后，在返回总的结果给调用者之前要加上最后一个块的结果❿。

因此，这就去除了一个可能的问题，工作线程中抛出的异常会在主线程中再次被抛出。如果多于一个工作线程抛出异常，只有一个异常会被传播，但是这也不是一个大问题。如果确实有关的话，可以使用类似 std::nested_exception 来捕捉所有的异常

然后抛出它。

如果在你产生第一个线程和你加入它们之间抛出异常的话，那么剩下的问题就是线程泄漏。最简单的方法就是捕获所有异常，并且将它们融合到调用 joinable()的线程中，然后再次抛出异常。

```
try
{
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        // ... as before
    }
    T last_result=accumulate_block()(block_start,last);

    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));
}
catch(...)
{
    for(unsigned long i=0;i<(num_thread-1);++i)
    {
        if(threads[i].joinable())
            thread[i].join();
    }
    throw;
}
```

现在它起作用了。所有线程将被联合起来，无论代码是如何离开块的。尽管如此，try-catch 块是令人讨厌的，并且你有复制代码。你将加入正常的控制流以及捕获块的线程中。复制代码不是一个好事情，因为这意味着需要改变更多的地方。我们在一个对象的析构函数中检查它，毕竟，这是 C++中惯用的清除资源的方法。下面是你的类。

```
class join_threads
{
    std::vector<std::thread>& threads;
public:
    explicit join_threads(std::vector<std::thread>& threads_):
        threads(threads_)
    {}
    ~join_threads()
    {
        for(unsigned long i=0;i<threads.size();++i)
        {
            if(threads[i].joinable())
                threads[i].join();
        }
    }
};
```

这与清单 2.3 中的 thread_guard 类是相似的，除了它扩展为适合所有线程。你可以如清单 8.4 所示简化代码。

**清单 8.4 std::accumulate 的异常安全并行版本**

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return init;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;

    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();

    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);

    unsigned long const block_size=length/num_threads;

    std::vector<std::future<T> > futures(num_threads-1);
    std::vector<std::thread> threads(num_threads-1);
    join_threads joiner(threads);                          <-❶

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        std::packaged_task<T(Iterator,Iterator)> task(
            accumulate_block<Iterator,T>());
        futures[i]=task.get_future();
        threads[i]=std::thread(std::move(task),block_start,block_end);
        block_start=block_end;
    }
    T last_result=accumulate_block()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get();              <-❷
    }
    result += last_result;
    return result;
}
```

一旦你创建了你的线程容器，也就创建了一个新类的实例 ❶ 来加入所有在退出的线程。你可以去除你的联合循环，只要你知道无论函数是否退出，这些线程都将被联合起来。注意调用 futures[i].get() ❷ 将被阻塞直到结果出来，因此在这一点并不需要明确地与线程融合起来。这与清单 8.2 中的原型不一样，在清单 8.2 中你必须与线程联合起来确保正确复制了结果向量。你不仅得到了异常安全代码，而且你的函数也更短了，因为将联合代码提取到你的新（可再用的）类中了。

## 2．STD::ASYNC()的异常安全

你已经知道了当处理线程时需要什么来实现异常安全，我们来看看使用 std::async()时需要做的同样的事情。你已经看到了，在这种情况下库为你处理这些线程，并且当 future 是就绪的时候，产生的任何线程都完成了。需要注意到关键事情就是异常安全，如果销毁 future 的时候没有等待它，析构函数将等待线程完成。这就避免了仍然在执行以及持有数据引用的泄漏线程的问题。清单 8.5 所示就是使用 std::async()的异常安全实现。

**清单 8.5　使用 std::async 的 std::accumulate 的异常安全并行版本**

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);   <-❶
    unsigned long const max_chunk_size=25;
    if(length<=max_chunk_size)
    {
        return std::accumulate(first,last,init);   <-❷
    }
    else
    {
    Iterator mid_point=first;
        std::advance(mid_point,length/2);   <-❸
        std::future<T> first_half_result=
            std::async(parallel_accumulate<Iterator,T>,   <-❹
                       first,mid_point,init);
        T second_half_result=parallel_accumulate(mid_point,last,T());   <-❺
        return first_half_result.get()+second_half_result;
    }                                                  ❻
}
```

这个版本使用递归将数据划分为块而不是重新计算将数据划分为块，但是它比之前的版本要简单一些，并且是异常安全的。如以前一样，你以计算序列长度开始 ❶，如果它比最大的块尺寸小的话，就直接调用 std::accumulate❷。如果它的元素比块尺寸大的话，就找到中点 ❸ 然后产生一个异步任务来处理前半部分 ❹。范围内的第二部分就用一个直接的递归调用来处理 ❺，然后将这两个块的结果累加 ❻。库确保了 std::async 调用使用了可获得的硬件线程，并且没有创造很多线程。一些“异步”调用将在调用 get()的时候被异步执行 ❻。

这种做法的好处在于它不仅可以利用硬件并发，而且它也是异常安全的。如果递归调用抛出异常 ❺，当异常传播时，调用 std::async❹ 创造的 future 就会被销毁。它会轮流等待异步线程结束，因此避免了悬挂线程。另一方面，如果异步调用抛出异常，就会被 future 捕捉，并且调用 get()❻ 将再次抛出异常。

当设计并发代码的时候还需要考虑哪些方面呢？让我们来看看可扩展性。如果将你的代码迁移到更多处理器系统中会提高多少性能呢？

## 8.4.2 可扩展性和阿姆达尔定律

可扩展性是关于确保你的应用可以利用系统中增加的处理器。一种极端情况就是你有一个完全不能扩展的单线程应用，即使你在系统中增加100个处理器也不会改变性能。另一种极端情况是你有类似SETI@Home[1]的项目，被设计用来利用成千上万的附加的处理器（以用户将个人计算机增加到网络中的形式）。

对于任何给定的多线程程序，当程序运行时，执行有用工作的线程的数量会发生变化。即使每个线程都在做有用的操作，初始化应用的时候可能只有一个线程，然后就有一个任务生成其他的线程。但是即使那样也是一个不太可能发生的方案。线程经常花时间等待彼此或者等待I/O操作完成。

除非每次线程等待事情（无论是什么事情）的时候都有另一个线程在处理器上代替它，否则就有一个可以进行有用工作的处理器处于闲置状态。

一种简单的方法就是将程序划分为只有一个线程在做有用的工作“串行的”部分和所有可以获得的处理器都在做有用工作的“并行的”部分。如果你在有更多处理器的系统上运行你的应用，理论上就可以更快地完成“并行”部分，因为可以在更多的处理器间划分工作，但是“串行的”部分仍然是串行的。在这样一种简单假设下，你可以通过增加处理器数量来估计可以获得的性能。如果“连续的”部分组成程序的一个部分 $f_s$，那么使用 $N$ 个处理器获得的性能 $P$ 就可以估计为

$$P = \frac{1}{f_s + \frac{1 - f_s}{N}}$$

这就是阿姆达尔定律（**Amdahl's law**），当谈论并发代码性能的时候经常被引用。如果所有事情都能被并行，那么串行部分就为0，加速就是 $N$。或者，如果串行部分是三分之一，即使有无限多的处理器，你也不会得到超过3的加速。

尽管如此，这是一种很理想的情况。因为任务很少可以像方程式所需要的那样被无穷划分，并且所有事情都达到它所假设的CPU界限是很少出现的。正如你看到的，线程执行的时候会等待很多事情。

阿姆达尔定律中有一点是明确的，那就是当你为性能使用并发的时候，值得考虑总体应用的设计来最大化并发的可能性，并且确保处理器始终有有用的工作来完成。如果你可以减少“串行”部分或者减少线程等待的可能性，你就可以提高在有更多处理器的

[1] http://setiathome.ssl.berkeley.edu/

系统上的性能。或者，如果你可以为系统提供更多的数据，并且保持并行部分准备工作，就可以减少串行部分，增加性能 $P$ 的值。

从根本上说，可扩展性就是当增加更多的处理器的时候，可以减少它执行操作的时间或者增加在一段时间内处理的数据数量。有时这两点是相同的（如果每个元素可以处理得更快，那么你就可以处理更多数据），但是并不总是一样的。在选择在线程间划分工作的方法之前，识别出可扩展性的哪些方面对你很重要是很必要的。

在这部分的开始我就提到过线程并不是总有有用的工作来做。有时它们必须等待别的线程，或者等待 I/O 操作完成，或者别的事情。如果在等待中你给系统一些有用的事情，你就可以有效的“隐藏”等待。

### 8.4.3 用多线程隐藏延迟

在很多关于多线程代码性能的讨论中，我们都假设当它们真正在处理器上运行时，线程在“全力以赴”的运行并且总是有有用的工作来做。这当然不是正确的，在应用代码中，线程在等待的时候总是频繁地被阻塞。例如，它们可能在等待一些 I/O 操作的完成，等待获得互斥元，等待另一个线程完成一些操作并且通知一个条件变量，或者只是休眠一段时间。

无论等待的原因是什么，如果你只有和系统中物理处理单元一样多的线程，那么有阻塞的线程就意味着你在浪费 CPU 时间。运行一个被阻塞的线程的处理器不做任何事情。因此，如果你知道一个线程将会有相当一部分时间在等待，那么你就可以通过运行一个或多个附加线程来使用那个空闲的 CPU 时间。

考虑一个病毒扫描应用，它使用管道在线程间划分工作。第一个线程搜索文件系统来检查文件并且将它们放到队列中。同时，另一个线程获得队列中的文件名，载入文件，并且扫描它们的病毒。你知道搜索文件系统的文件来扫描的线程肯定会达到 I/O 界限，因此你就可以通过运行一个附加的扫描线程来使用“空闲的”CPU 时间。那么你就有一个搜索文件线程，以及与系统中的物理核或者处理器相同数量的扫描线程。因为扫描线程可能也不得不从磁盘读取文件的重要部分来扫描它们，拥有更多扫描线程也是很有意义的。但是在某个时刻会有太多线程，系统会再次慢下来因为它花了更多时间切换程序，正如 8.2.5 节所描述的。

仍然，这是一个最优化问题，因此测量线程数量改变前后的性能时很重要的；最有的线程数量将很大程度上取决于工作的性质和线程等待的时间所占的比例。

取决于应用，它也可能用完空闲的 CPU 时间而没有运行附加的线程。例如，如果一个线程因为等待 I/O 操作的完成而被阻塞，那么使用可获得的异步 I/O 操作就很有意义了。那么当在背后执行 I/O 操作的时候，线程就可以执行别的有用的工作了。在别的情况下，如果一个线程在等待另一个线程执行一个操作，那么等待的线程就可以自己执行那个操作而不是被阻塞，正如第 7 章中的无锁队列。在一个极端的情况下，如果线程等待完成一个任务并且该任务没有被其他线程执行，等待的线程可以在它内部或者另一

个未完成的任务中执行这个任务。清单 8.1 中你看到了这个例子，在排序程序中只要它需要的块没有排好序就不停地排序它。

有时它增加线程来确保外部事件及时被处理来增加系统响应性，而不是增加线程来确保所有可得到的处理器都被应用了。

### 8.4.4 用并发提高响应性

很多现代图形用户接口框架是事件驱动的，使用者通过键盘输入或者移动鼠标在用户接口上执行操作，这会产生一系列的事件或者消息，稍后应用就会处理它。系统自己也会产生消息或者事件。为了确保所有事件和消息都能被正确处理，通常应用都有下面所示的一个事件循环。

```
while(true)
{
    event_data event=get_event();
    if(event.type==quit)
        break;
    process(event);
}
```

显然，API 的细节是不同的，但是结构通常是一样的，等待一个事件，做需要的操作来处理它，然后等待下一个事件。如果你有单线程应用，就会导致长时间运行的任务很难被写入，如 8.1.3 节描述的。为了确保用户输入能被及时处理，`get_event()`和`process()`必须以合理的频率被调用，无论应用在做什么操作。这就意味着要么任务必须定期暂停并且将控制交给事件循环，要么方便的时候在代码中调用`get_event()/process()`代码。每一种选择都将任务的实现变得复杂了。

通过用并发分离关注点，你就可以将长任务在一个新线程上执行，并且用一个专用的 GUI 线程来处理事件。线程可以通过简单的方法互相访问，而不是必须以某种方式将事件处理代码放到任务代码中。清单 8.6 列出了这种分离的简单概括。

**清单 8.6 从任务线程中分离 GUI 线程**

```
std::thread task_thread;
std::atomic<bool> task_cancelled(false);
void gui_thread()
{
    while(true)
    {
        event_data event=get_event();
        if(event.type==quit)
            break;
        process(event);
    }
}
```

```
void task()
{
    while(!task_complete() && !task_cancelled)
    {
        do_next_operation();
    }
    if(task_cancelled)
    {
        perform_cleanup();
    }
    else
    {
        post_gui_event(task_complete);
    }
}

void process(event_data const& event)
{
    switch(event.type)
    {
    case start_task:
        task_cancelled=false;
        task_thread=std::thread(task);
        break;
    case stop_task:
        task_cancelled=true;
        task_thread.join();
        break;
    case task_complete:
        task_thread.join();
        display_results();
        break;
    default:
        //...
    }
}
```

通过这种方式分离障碍，用户线程能够及时地响应事件，即使任务要执行很长时间。这种响应性通常是使用应用时用户体验的关键。无论何时执行一个特定操作（无论该操作是什么），应用都会被完全锁住，这样使用起来就不方便了。通过提供一个专用的事件处理线程，GUI 可以处理 GUI 特有的消息（例如调整窗口大小或者重画窗口）而不会中断耗时处理的执行，并且当它们确实影响长任务时会传递相关的消息。

本章你看到了设计并发代码的时候需要考虑的问题。就整体而言，这些问题是很大的，但是当你习惯了“多线程编程”，它们就会变得得心应手了。如果这些考虑对你来说很新，那么希望当你看到它们是如何影响多线程代码的具体例子的时候，可以变得更清晰。

# 8.5 在实践中设计并发代码

当为一个特别的任务设计并发代码的时候，你需要事先考虑每个描述的问题的程度将取决于任务。为了演示它们是如何应用的，我们将看看 C++标准库中三个函数并行版本的实现。这将给你一个类似的构造基础，提供了考虑问题的平台。作为奖励，我们也有可用的函数实现，可用来帮助并行一个更大的任务。

我选择这些实现主要是用来演示特别的方法而不是成为最高水平的实现。在并行算法学术文献或者在多线程库例如 Inter's Threading Building Blocks[1]中可以找到更高程度的实现，它们更好地利用了可获得的硬件并发性。

概念上最简单的并行算法是 std::for_each 的并行版本，我们就先从它开始。

## 8.5.1 std::for_each 的并行实现

std::for_each 在概念上很简单，它轮流在范围内的每个元素上调用用户提供的函数。std::for_each 的并行实现和串行实现的最大区别就是函数调用的顺序。std::for_each 用范围内的第一个元素调用函数，然后是第二个，以此类推。然而使用并行实现就不能保证元素被处理的顺序，它们可能（实际上我们希望）被并发处理。

为了实现并行版本，你只需要将这个范围划分为元素集合在每个线程上处理。你事先知道了元素数量，因此你可以在处理开始前划分数据（参见 8.1.1 节）。我们假设这是唯一在运行的并行任务，那么就可以使用 std::thread::hardware_concurrency()来决定线程的数量。你也知道元素可以被独立地处理，因此你可以使用临近的块来避免假共享（参见 8.2.3 节）。

这个算法与 8.4.1 节中描述的 std::accumulate 的并行版本在概念上是类似的，但是不计算每个元素的和，你只要应用具体函数。尽管你可能推测这会大大简化代码，因为它不返回结果。如果你想传递异常给调用者，你仍然需要使用 std::packaged_task 和 std::future 方法在线程间传递异常。一个简单的实现如清单 8.7 所示。

**清单 8.7 std::for_each 的并行版本**

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
```

[1] http://threadingbuildingblocks.org/

```
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;
    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();

    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);

    unsigned long const block_size=length/num_threads;

    std::vector<std::future<void> > futures(num_threads-1);      ←❶
    std::vector<std::thread> threads(num_threads-1);
    join_threads joiner(threads);

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        std::packaged_task<void(void)> task(      ←❷
            [=]()
            {
                std::for_each(block_start,block_end,f);
            });
        futures[i]=task.get_future();
        threads[i]=std::thread(std::move(task));      ←❸
        block_start=block_end;
    }
    std::for_each(block_start,last,f);
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        futures[i].get();      ←❹
    }
}
```

这段代码的基础结构与清单 8.4 中的代码是一样的。关键的不同之处在于 futures 向量存储了 std::future<void>❶，因为工作线程不返回值，并且在这个任务上使用一个简单 lambda 函数激活了从 block_start 到 block_end 范围上的函数 f❷。这就避免了将范围传递给线程构造函数❸。因为工作线程不返回值，调用 future[i].get()❹只提供取回工作线程抛出的异常的方法，如果你不希望传递异常，那么你就可以省略它。

正如你的 std::accumulate 的并行实现可以通过使用 std::async 被简化，因此你的 parallel_for_each 也可以被简化。它的实现如清单 8.8 所示。

清单 8.8 使用 std::async 的 std::for_each 的并行版本

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return;

    unsigned long const min_per_thread=25;

    if(length<(2*min_per_thread))
    {
        std::for_each(first,last,f);     <-❶
    }
    else
    {
        Iterator const mid_point=first+length/2;
        std::future<void> first_half=                        <-❷
            std::async(&parallel_for_each<Iterator,Func>,
                       first,mid_point,f);
        parallel_for_each(mid_point,last,f);       <-❸
        first_half.get();     <-❹
    }
}
```

如同清单 8.5 中基于 std::async 的 parallel_accumulate 一样，你递归地划分数据而不是在执行前划分数据，因为你不知道库会使用多少线程。如以前一样，每一步都将数据划分为两部分，异步运行前半部分 ❷ 并且直接运行后半部分 ❸ 直到剩下的数据太小而不值得划分，在这种情况下会调用 std::for_each❶。使用 std::async 和 get() 成员函数 std::future❹ 提供了异常传播语义。

让我们从必须在每个元素上执行相同操作的算法转移到稍微复杂的例子 std::find。

## 8.5.2 std::find 的并行实现

std::find 是下一个考虑的有用的算法，因为它是不用处理完所有元素就可以完成的几个算法之一。例如，如果范围内的第一个元素符合搜索准则，那么就不需要检查别的元素。稍后你将看到，这是性能的一个重要属性，并且对设计并行实现有直接影响。这是数据读取部分可能影响代码设计的一个特殊例子（参见 8.3.2 节）。这类别的算法包括 std::equal 和 std::any_of。

如果你和你的妻子或者搭档在阁楼的两箱纪念品中寻找一张旧相片，如果你找到了相片就应该让他们也停止寻找。你要让他们知道你已经找到了相片（可以通过呼喊，“找到了”），这样他们就可以停止寻找并且做别的事情。很多算法的天性是处理每个元素，因此它们没有呼喊“找到了”。对于算法例如 std::find，早日完成的能力是一个重要

的特性并且不浪费任何事情。因此你需要设计你的代码来使用这个特性——当知道结果的时候用一些方式中断别的任务，因此代码不需要等待别的工作线程处理剩下的元素。

如果你不中断别的线程，串行版本比并行版本的性能更好，因为串行算法一旦找到匹配项就停止搜索并且返回。例如，如果系统可以支持四个并发线程，每个线程将检查范围内四分之一的元素，并且我们的并行算法大约花费单个线程四分之一的时间来检查每个元素。如果匹配的元素位于范围内的前四分之一，串行算法会首先返回，因为它不需要检查剩下的元素。

你可以中断别的线程，一种方法是通过使用一个原子变量作为一个标志，并且在处理完每个元素后检查这个标志。如果设置了标志，就代表有一个线程发现了匹配项，因此就可以终止执行并且返回了。用这种方法中断别的线程，你保持了你不需要处理每个值的特性，因此在更多的情况下与串行版本相比提高了性能。这种方法的缺点是原子载入变成慢动作，这就会妨碍每个线程的前进。

关于如何返回值和如何传递异常你有两个选择。你可以使用 future 数组，使用 std::packaged_task 来转移值和异常，然后在主线程中处理返回的结果；或者你使用 std::promise 来从工作线程中直接设置最终结果。如果你希望在第一个异常处停止（即使你没有处理完所有元素），你可以使用 std::promise 来设置值和异常。另一方面，如果你希望允许别的工作线程继续搜索，你可以使用 std::packaged_task，存储所有的异常，那么没有发现匹配项的时候就重新抛出其中之一。

在这种情况下，我选择使用 std::promise，因为行为更匹配 std::find。这里要注意的一件事情就是要搜索的元素不在提供的范围内。因此在从 future 得到结果之前你需要等待所有线程结束。如果你在 future 上阻塞了并且没有这个值的话，你就会永远等待。结果如清单 8.9 所示。

**清单 8.9　并行 find 算法的一种实现**

```
template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
    struct find_element     <—❶
    {
    void operator()(Iterator begin,Iterator end,
                    MatchType match,
                    std::promise<Iterator>* result,
                    std::atomic<bool>* done_flag)
    {
        try
        {
            for(;(begin!=end) && !done_flag->load();++begin)     <—❷
            {
                if(*begin==match)
                {
```

```
                result->set_value(begin);          <-3
                done_flag->store(true);          <-4
                return;
            }
        }
    }
    catch(...)         <-5
    {
        try
        {
            result->set_exception(std::current_exception());   <-6
            done_flag->store(true);
        }
        catch(...)    <-7
        {}
    }
}
};
unsigned long const length=std::distance(first,last);

if(!length)
    return last;

unsigned long const min_per_thread=25;
unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);

unsigned long const block_size=length/num_threads;

std::promise<Iterator> result;   <-8
std::atomic<bool> done_flag(false);   <-9
std::vector<std::thread> threads(num_threads-1);
{                                          <-10
    join_threads joiner(threads);

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);

        threads[i]=std::thread(find_element(),             <-11
                               block_start,block_end,match,
                               &result,&done_flag);
        block_start=block_end;
    }
    find_element()(block_start,last,match,&result,&done_flag);   <-12
}
if(!done_flag.load())   <-13
{
    return last;
```

```
    }
    return result.get_future().get();  ⟵⑭
}
```

清单 8.9 的主函数与之前的例子很相似。这次，在本地 find_element 类的函数调用操作上完成工作 ❶。这个循环访问给定的块的每个元素，每一步都检查标志 ❷。如果找到匹配项，就在 promise 中设置最后的结果 ❸ 并且在返回前设置 done_flag❹。

如果抛出异常，就可以被捕获 ❺，并且在设置 done_flag 前在 promise 中存储异常 ❻。如果 promise 已经被设置了，再设置值就有可能抛出异常，因此你捕获并且抛弃发生在这里的异常❼。

这就意味着如果一个线程调用 find_element，要么找到匹配项要么抛出异常，此时所有别的线程都会看到 done_flag 被设置了并且停止执行。如果多个线程同时找到匹配项或者抛出异常，它们就会在设置 promise 值的时候产生竞争。但是这是一个没有危害的竞争条件，无论哪一个线程成功都会成为名义上的“第一个”并且是一个可接受的结果。

回顾主函数 parallel_find 本身，你用 promise❽和 flag❾来停止搜索，这两者都传递给在这个范围内搜索的新线程⓫。主线程也使用 find_element 来搜索剩下的元素⓬。我们已经说过，你在检查结果之前需要等待所有线程结束，因为可能没有匹配的元素。你通过在块中附入线程连接的代码来完成❿，因此当你检查标志来看看是否发现匹配项的时候，所有线程都被联合起来了⓭。如果发现匹配项，你就可以通过在 std::future<iterator>中调用 get()得到结果或者抛出存储的异常⓮。

同样，这个实现假设你将使用所有可获得的硬件线程或者你有别的方法来决定线程数量，用来提前在线程间划分工作。跟以前一样，在使用 C++标准库的自动扩展功能的时候，你可以使用 std::async 和递归数据划分来简化你的实现。使用 std::async 的 parallel_find 实现如清单 8.10 所示。

**清单 8.10　使用 std::async 的并行查找算法的实现**

```
template<typename Iterator,typename MatchType>                      ⟵❶
Iterator parallel_find_impl(Iterator first,Iterator last,MatchType match,
                            std::atomic<bool>& done)
{
    try
    {
        unsigned long const length=std::distance(first,last);
        unsigned long const min_per_thread=25;               ⟵❷
        if(length<(2*min_per_thread))                ⟵❸
        {
            for(;(first!=last) && !done.load();++first)          ⟵❹
            {
                if(*first==match)
                {
```

```
                done=true;    ⑤
                return first;
            }
        }
        return last;    ⑥
    }
    else
    {
        Iterator const mid_point=first+(length/2);    ⑦
        std::future<Iterator> async_result=
            std::async(&parallel_find_impl<Iterator,MatchType>,    ⑧
                       mid_point,last,match,std::ref(done));
        Iterator const direct_result=
                parallel_find_impl(first,mid_point,match,done);    ⑨
        return (direct_result==mid_point)?
            async_result.get():direct_result;    ⑩
    }
}
catch(...)
{
    done=true;    ⑪
    throw;
}
}

template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
    std::atomic<bool> done(false);
    return parallel_find_impl(first,last,match,done);    ⑫
}
```

如果你找到匹配项就结束查找，意味着你需要引入一个在线程间共享的标志，用来表示已经找到该匹配项。因此这就需要传递给所有递归调用。实现它最简单的方法就是通过在实现函数上 ❶ 附加参数——引用 done 标志，这是从主入口点传递进来的⓬。

核心实现在类似的代码行里继续执行。与很多实现相同，你在单线程上设置处理项的最小值 ❷。如果你不能将它划分为都至少达到设置的大小的两部分，就在当前线程上运行所有的事情 ❸。实际算法是处理具体范围的简单循环，一直循环直到范围结束或者设置了 done 标志 ❹。如果你查找到匹配项，就在返回前设置 done 标志 ❺。如果你停止查找，要么因为你已经到达范围的末端，要么因为另一个线程设置了 done 标志。你返回 last 用来表示在这里没有匹配项 ❻。

如果范围可以被划分，你在使用 std::async 前首先发现中点❼，以便在范围的后半部分进行查找❽，小心使用 std::ref 来传递 done 标志的引用。同时，你可以通过直接递归调用在范围的前半部分进行查找❾。如果原来的范围太大的话，这个异步调用和直接递归可能导致进一步的划分。

如果直接搜索返回 mid_point，那么就没有找到匹配项，你就需要得到异步搜索

的结果。如果那部分没有发现结果，结果就将是 last，这是正确的返回值表明没有查找到这个值❿。如果“异步的”调用被延迟而不是真正的异步，它在调用 get()的时候才真正运行，在这种情况下，如果在后半部分查找到了就跳过搜索范围的前半部分。如果异步搜索已经在另一个线程上运行了，async_result 变量的析构函数将等待线程完成。这样就不会有任何泄露线程。

如同以前一样，使用 std::async 提供了异常安全和异常传递特性。如果直接递归抛出异常，future 的析构函数将确保运行异步调用的线程在函数返回前就结束了。并且如果异步调用抛出异常，这个异常就通过 get()调用传递❿。使用一个 try/catch 块是为了在异常上设置 done 标志并且确保如果抛出异常的话所有线程很快终止⓫。没有它，实现仍然是正确的，但是会继续检查元素直到每个线程都结束了。

这个算法的两种实现与另一种并行算法共享的主要特点就是不再保证项目按照从 std::find 得到的顺序来处理。如果你想并行算法这一点是很基础的。如果顺序很重要的话，你就不能并发处理元素。如果元素是独立的，就可以使用 parallel_for_each。但是这意味着你的 parallel_find 可能返回范围尾部的元素，即使它与范围头部的元素匹配。

好了，你已经处理了并行化 std::find。正如我在这一节开头说的那样，存在别的类似算法可以在不处理每个数据元素的情况下完成，并且可以使用同样的方法。我们将在第 9 章中进一步讨论中断线程的问题。

为了完成我们的例子，我们将从不同的方面来考虑并且看看 std::partial_sum。这个算法是一个很有趣的并行算法并且强调了一些附加的设计选择。

### 8.5.3 std::partial_sum 的并行实现

std::partial_sum 计算了一个范围内的总和，因此每个元素都被这个元素与它之前的所有元素的和所代替。所以序列 1、2、3、4、5 就变成 1、(1+2)=3、(1+2+3)=6、(1+2+3+4)=10、(1+2+3+4+5)=15。它的并行化是很有趣的，因为你不能将范围划分为块然后单独计算每个块。例如，第一个元素的原始值需要加到每一个别的元素上。

一种用来决定范围内部分和的方法就是计算独立块的部分和，然后将第一个块中计算得到的最后一个元素的值加到下一个块的元素中，并以此类推。如果你有元素 1,2,3,4,5,6,7,8,9 并且你将它分成三个块，首先你得到{1,3,6}、{4,9,15}、{7,15,24}。如果你将 6 加到第二个块的所有元素上，你就得到{1,3,6}、{10,15,21}、{7,15,24}。然后你将第二个块的最后一个元素{21}加到第三个块的元素上，这样最后一个块得到最后的值：{1,3,6}、{10,15,21}、{28,36,55}。

同原始划分成块一样，也可以并行加上前一个块的部分和。如果每个块的最后一个元素首先被更新，那么当第二个线程更新下一个块的时候，第一个线程可以更新这个块

中剩下的元素，并且以此类推。当列表中的元素多于处理核的时候，这种方法很有效，因为每个核每一步都要处理大量元素。

如果你有很多处理核（同元素数量一样，或者多于元素数量），这种方法就不是很有效了。如果你在处理器间划分工作，第一步以元素对结束工作。这这种条件下，这种进一步传递结果意味着很多处理器会等待，因此你需要给它们分配工作。你可以采用另一种方法对待这个问题。你做部分传递，而不是从一个块到下一个块做全部和的传递。首先像之前一样求和相邻的元素，然后将这些和加到与它相隔两个的元素上，然后再将得到的值加到与它相隔四个的元素上，以此类推。如果你以同样的九个元素开始，第一轮后你得到 1,3,5,7,9,11,13,15,17，这就得到前两个元素最后的值。第二轮后你得到 1,3,6,10,14,18,22,26,30，它的前四个元素是正确的。第三轮后你得到 1,3,6,10,15,21,18,36,44，它的前八个元素是正确的。第四轮后你得到 1,3,6,10,15,21,18,36,45，这就是最终答案。尽管它比第一种方法的总步骤数要多，但是如果你有很多线程的话，它有更大的余地来并行化，每个处理器每一步都能更新一个入口。

总的说来，第二种方法执行 $\log_2(N)$个步骤，每一个步骤执行大约 $N$ 个操作（每个线程处理一个），其中 $N$ 是表中的元素数量。第一种算法中，每个线程为分配给它的块的最初分段求和执行 $\frac{N}{k}$ 个操作，然后为进一步的传递执行 $\frac{N}{k}$ 个操作，$k$ 是线程数量。因此从总的操作数来说，第一种方法是 $O(N)$，第二种方法是 $O(N\log(N))$。尽管如此，如果你的处理器与列表中的元素一样多，那么第二种方法中**每个处理器**只需要 $\log(N)$个操作。而当 $k$ 很大时，第一种方法本质上是串行操作，因为需要进一步传递值。对于拥有很少处理单元的系统，第一种方法可以更快结束，而对于大规模并行系统，第二种方法可以更快结束。8.2.1 节讨论了这个问题的一个极端的例子。

无论如何，先不考虑效率问题，我们来看一些代码。清单 8.11 所示的是第一种方法。

**清单 8.11　通过划分问题来并行计算分段的和**

```
template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
   typedef typename Iterator::value_type value_type;

   struct process_chunk        <-❶
   {
       void operator()(Iterator begin,Iterator last,
                       std::future<value_type>* previous_end_value,
                       std::promise<value_type>* end_value)
       {
           try
           {
               Iterator end=last;
               ++end;
```

```
            std::partial_sum(begin,end,begin);          ❷
            if(previous_end_value)                  ❸
            {
                value_type& addend=previous_end_value->get();   ❹
                *last+=addend;                  ❺
                if(end_value)
                {
                    end_value->set_value(*last);        ❻
                }
                std::for_each(begin,last,[addend](value_type& item)   ❼
                              {
                                  item+=addend;
                              });
            }
            else if(end_value)
            {
                end_value->set_value(*last);        ❽
            }
        }
        catch(...)       ❾
        {
            if(end_value)
            {
                end_value->set_exception(std::current_exception());   ❿
            }
            else
            {
                throw;       ⓫
            }
        }
    }
};

unsigned long const length=std::distance(first,last);

if(!length)
    return last;

unsigned long const min_per_thread=25;       ⓬
unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;

unsigned long const hardware_threads=
    std::thread::hardware_concurrency();

unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);

unsigned long const block_size=length/num_threads;

typedef typename Iterator::value_type value_type;

std::vector<std::thread> threads(num_threads-1);       ⓭
std::vector<std::promise<value_type> >
    end_values(num_threads-1);       ⓮
std::vector<std::future<value_type> >
```

```
        previous_end_values;        ← ⑮
    previous_end_values.reserve(num_threads-1);    ← ⑯
    join_threads joiner(threads);

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_last=block_start;
        std::advance(block_last,block_size-1);    ← ⑰
        threads[i]=std::thread(process_chunk(),    ← ⑱
                               block_start,block_last,
                               (i!=0)?&previous_end_values[i-1]:0,
                               &end_values[i]);
        block_start=block_last;        ⑲
        ++block_start;                 ←
        previous_end_values.push_back(end_values[i].get_future());    ← ⑳
    }
    Iterator final_element=block_start;
    std::advance(final_element,std::distance(block_start,last)-1);    ← ㉑
    process_chunk()(block_start,final_element,    ← ㉒
                    (num_threads>1)?&previous_end_values.back():0,
                    0);
}
```

在这个例子中，总体结构与之前的代码是一样的，划分问题为块，每个线程拥有最小化的块尺寸⑫。在这种情况下，与线程向量一样⑬，你有一个 promise 向量⑭用来存储块中最后一个元素的值，并且还有一个 future 向量⑮，用来得到前一个块的最后一个值。你可以保留 future 的空间⑯来避免在生成线程的时候再分配，因为你知道将有多少线程。

主循环与以前的一样，只是这次你希望迭代器指向每个块中的最后一个元素本身，而不是通常情况下那样指向最后元素的后继⑰，因此在每个范围你都可以传递最后一个元素。真正的处理是在 process_chunk 函数对象中完成的，我们稍后再分析。需要给的参数包括这个块的开始和结束的迭代器，先前范围的终值（如果存在的话），这个范围的终值存放的位置⑱。

产生线程后，你就可以更新这个块的起点，记住将它的值加一使它位于最后一个元素之后⑲，并且将当前块的最后一个值的 future 存储到 future 向量中，这样下一次循环的时候就可以得到它⑳。

在你处理最后一个块之前，你需要得到最后一个元素的迭代器㉑，这样你就可以传递到 process_chunk 中㉒。std::partial sum 不返回值，因此一旦最后一个块被处理了，你不需要做任何操作。当所有线程结束的时候这个操作就完成了。

现在我们来看看 process_chunk 函数对象在整个工作中所起的作用 ❶。首先在整个块上调用 std::partial_sum，包括最后一个元素 ❷，但是然后就需要知道这是否为第一个块 ❸。如果这不是第一个块，那么在先前块中就有 previous_end_value，因此你就需要等待这个值 ❹。为了最大化算法的并发性，你就需要首先更新最后一个元素 ❺，因此你将值传递给下一个块（如果存在的话）❻。一旦完成了这些操作，你就可以使用 std::for_each 和一个简单的 lambda 函数❼来更新这个范围内剩下的元素。

如果没有 previous_end_value，那么这就是第一个块，因此你可以只更新下一个块的 end_value()（同样，如果存在下一个块的话——这可能是仅有的块）❽。

最后，如果任何操作抛出异常，你捕捉到它❾并且将它存储在 promise 中❿。这样当它试图得到先前的最后一个值的时候就会传递给下一个块了❹。这会将所有的异常都传递给最后一个块，然后被重新抛出⓫，因为你知道这是运行在主线程上。

因为线程间的同步，这段代码就不容易控制与 std::async 一起重写。等待结果的任务中途通过别的任务的执行，因此所有任务都必须同时运行。

使用基于块，向前传输的方法是很少见的，让我们来看看计算范围内部分和的第二种方法。

### 为部分求和实现增量对偶算法

当你的处理器可以按部就班地执行加法，通过累加越来越远的元素来计算部分和的方法可以工作得很好。在这种情况下，不需要进一步的同步，因为所有中间结果都会直接传递给下一个需要它们的处理器。但是实际上，你很少可以在这样的系统上工作，除非你的系统支持少量数据元素上同时执行操作的指令，即所谓的单指令/多数据（Single-Instruction/Multiple-Data，SIMD)指令。因此，你必须为通常情况设计代码并且每一步都明确地同步线程。

一种方法是使用屏障（**barrier**）——一种同步方法使得线程等待直到要求的线程已经到达了屏障。一旦所有线程到达屏障，它们就可以继续执行而不阻塞了。C++11 线程库没有直接提供这样的工具，因此你需要自己设计。

想象一下游乐园里的过山车。如果有合适数量的人在等待，游乐园职员就会确保在过山车离开平台之前每个座位都有人。屏障的工作原理也是一样的，你提前指定“座位”的数量，并且线程必须等待直到所有“座位”都是满的。一旦有足够的等待线程，就可以继续执行，屏障被重置并且开始等待下一批线程。通常，在循环中使用此构造，在那里同一线程再次出现并且等待下一次。这个想法是为了使线程按部就班地工作，因此一个线程不会在别的线程前面离开以及掉队。对于这个算法，这就会造成灾难，因为离开的线程可能会修改别的线程正在使用的数据，或者使用还没有被正确更新的数据。

无论如何，清单 8.12 给出了屏障的一个简单实现。

**清单 8.12 一个简单的屏障类**

```
class barrier
{
    unsigned const count;
    std::atomic<unsigned> spaces;
    std::atomic<unsigned> generation;
public:
    explicit barrier(unsigned count_):    ❶
```

```
        count(count_),spaces(count),generation(0)
    {}
    void wait()
    {
        unsigned const my_generation=generation;   ←②
        if(!--spaces)                               ←③
        {
            spaces=count;           ←④
            ++generation;           ←⑤
        }
        else
        {
            while(generation==my_generation)        ←⑥
                std::this_thread::yield();          ←⑦
        }
    }
};
```

在这个实现中，你构造了一个 barrier，它具有一定数量"座位"❶，存储在 count 变量中。最初，屏障中 spaces 的数量与 count 相等。当每个线程等待的时候，spaces 的数量就会减 1❸。当它的值为零的时候，spaces 的值就会重置为 count❹，并且 generation 的值增一来给别的线程发信号表示它们可以继续执行❺。如果空闲 spaces 的数量没有达到零，你就必须等待。这个实现使用一种简单旋转锁❻，检查在 wait()开始的时候得到的值。因为当所有线程到达屏障的时候才会更新 generation 的值❺，等待的时候调用 yield()❼，因此在一个繁忙的等待中，等待的线程不会独占 CPU。

当我说这个实现是简单的，我的意思是它使用一个旋转锁，因此它不适合线程可能等待很长时间的情况。并且如果在任何一个时间有多于 count 数量的线程可能调用 wait()，那么它就不起作用了。如果你想处理这些情况中的任何一种，你就必须使用一个更健壮性（但是更复杂）的实现来代替。我遵循了在原子变量上的顺序连续操作，因为这使得事情更容易解释，但是你可以放松一些顺序约束。在大规模并发结构上，这样的全局同步是代价很高的，因为缓存线持有屏障的状态必须在所有涉及到的线程间穿梭（参见 8.2.2 节），因此你必须注意确保在这里这是最好的选择。

无论如何，在这里这就是你所需要的，你有固定数量的线程在循规蹈矩的循环中运行。当然，这只是几乎固定数量的线程。你可能记得，在一些步骤后，列表开始的项获得它的最终值。这就意味着要么你保持这些线程循环直到处理完整个范围，要么你需要允许屏障处理线程退出并且减少 count。我倾向于后面一种选择，因为它避免了使线程只是循环而不做任何有用的工作直到最后一步完成。

这就意味着你需要将 count 变成一个原子变量，因此你可以从多个线程更新它而不需要额外的同步。

```
std::atomic<unsigned> count;
```

它的初始化是一样的，但是现在当你重置 spaces 的值的时候就必须明确在 count

中 load()。

```
spaces=count.load();
```

这些都是在 wait() 上所做的改变；现在你需要一个新的成员函数来减少 count。我们称之为 done_waiting()，因为一个线程声称在等待的时候完成它。

```
void done_waiting()
{
    --count;                     <-❶
    if(!--spaces)                <-❷
    {
        spaces=count.load();     <-❸
        ++generation;
    }
}
```

你做的第一件事是递减 count 的值 ❶，这样下一次重置 spaces 就反映新的等待线程的数量。然后你需要将空闲 spaces 的数目递减 ❷。如果不这么做，别的线程就会永远等待，因为 spaces 被初始化为旧的，更大的值。如果这是分支上的最后一个线程，你就需要重置 counter 并且增加 generation❸，就如你在 wait() 中所做的那样。在完成所有的等待后就结束了。

现在你准备好写部分和的第二种实现了。在每一步，每个线程在屏障上调用 wait() 来保证线程步骤一起，并且一旦每个线程都完成了，就在屏障上调用 done_waiting() 来减少 count。如果你在初始范围内使用第二个缓冲器，屏障提供你所需要的所有同步。在每一步中，线程从原先的范围或者缓冲器中读取，并且将新值写入相关元素中。如果线程在一个步骤中读取了原先的范围，它们在下一个步骤中读取缓冲器，反之亦然。这就确保了在不同线程的读取和写操作中没有竞争条件。一旦一个线程结束循环，它必须确保正确的最终值被写入最初的范围中。清单 8.13 给出了所有的操作。

**清单 8.13　通过成对更新的 partial_sum 的并行实现**

```
struct barrier
{
    std::atomic<unsigned> count;
    std::atomic<unsigned> spaces;
    std::atomic<unsigned> generation;

    barrier(unsigned count_):
        count(count_),spaces(count_),generation(0)
    {}

    void wait()
    {
        unsigned const gen=generation.load();
        if(!--spaces)
        {
```

```
            spaces=count.load();
            ++generation;
        }
        else
        {
            while(generation.load()==gen)
            {
                std::this_thread::yield();
            }
        }
    }
    void done_waiting()
    {
        --count;
        if(!--spaces)
        {
            spaces=count.load();
            ++generation;
        }
    }
};
template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
    typedef typename Iterator::value_type value_type;
    struct process_element          <-❶
    {
        void operator()(Iterator first,Iterator last,
                        std::vector<value_type>& buffer,
                        unsigned i,barrier& b)
        {
            value_type& ith_element=*(first+i);
            bool update_source=false;
            for(unsigned step=0,stride=1;stride<=i;++step,stride*=2)
            {
                value_type const& source=(step%2)?       <-❷
                    buffer[i]:ith_element;
                value_type& dest=(step%2)?
                    ith_element:buffer[i];
                value_type const& addend=(step%2)?       <-❸
                    buffer[i-stride]:*(first+i-stride);
                dest=source+addend;     <-❹
                update_source=!(step%2);
                b.wait();               <-❺
            }
            if(update_source)           <-❻
            {
                ith_element=buffer[i];
            }
            b.done_waiting();           <-❼
        }
    };
```

```
    unsigned long const length=std::distance(first,last);
    if(length<=1)
        return;

    std::vector<value_type> buffer(length);
    barrier b(length);

    std::vector<std::thread> threads(length-1);    <-8
    join_threads joiner(threads);

    Iterator block_start=first;
    for(unsigned long i=0;i<(length-1);++i)
    {
        threads[i]=std::thread(process_element(),first,last,    <-9
                               std::ref(buffer),i,std::ref(b));
    }
    process_element()(first,last,buffer,length-1,b);    <-10
}
```

现在你已经很熟悉这段代码的总体结构了。你用一个拥有函数调用操作的类(process_element）来做这个工作 ❶，你在存储在向量中❽的一些线程❾上运行它并且在主线程调用它❿。这次的主要不同之处在于线程数量取决于列表中项的数量而不是取决于 std::thread::hardware_concurrency。正如我所说，除非你是在一个线程很便宜的大量并行机器上运行，这不是一个好方法，但是它显示了总体结构。也可以用较少的线程，每个线程处理源范围的多个值。但是这也会出现一个问题，那就是有足够少的线程使得它比向前传输算法的效率还要低。

无论如何，在 process_element 函数调用操作中完成了关键工作。每一步你要么从原先的范围得到第 i 个元素要么从缓冲器得到第 i 个元素 ❷，并且将它添加到 stride 元素的值中 ❸，如果从原先的范围开始就将它存储在缓冲器中，或者如果从缓冲器开始就存储在原先的范围中 ❹。然后在开始下一步之前你在屏障上等待 ❺。当 stride 使你离开范围的起点时你就完成操作了。在这种情况下，如果你的最终结果存储在缓冲器中的话，就更新原先范围里的元素 ❻。最后，你告诉屏障你正在 done_waiting() ❼。

注意这种情况不是异常安全的。如果一个工作线程在 process_element 上抛出异常，就会终止此引用。你可以使用 std::promise 存储异常来处理这种情况，正如你在清单 8.9 的 parallel_find 实现中所做的一样，或者使用互斥元保护的 std::exception_ptr。

这总结了我们的三个例子，希望它们有助于具体化 8.1，8.2 和 8.3 中强调的设计考虑，并且证明在真实代码中可以使用这些方法。

## 8.6 总结

本章我们涉及很多基础知识。我们从在线程间划分工作的多种方法开始，如提前划

分数据或者使用许多线程来形成管道。然后我们从低水平角度考虑与多线程代码的性能紧密相关的问题，在转移到数据读取是如何影响代码性能之前考虑了假共享和数据竞争问题。然后我们考虑了设计并发代码时候要考虑的问题，如异常安全和可扩展性。最后，我们以一些并发算法实现的例子作为结束，每一个例子都强调了当设计多线程代码时可能出现的具体问题。

本章出现几次的一个事情就是线程池的想法——一组事先配置的线程运行分配给线程池的任务。很多想法被用在设计一个好的线程池上，因此下一章我们将考虑一些问题，以及高级线程管理的别的方面。

# 第 9 章　高级线程管理

本章主要内容

- 线程池
- 处理线程池任务间的依赖
- 池中线程的工作窃取
- 中断线程

在之前的章节中，我们通过为每个线程创建 std::thread 对象来显式管理线程。在一些应用场合，你会发现这不是你所需要的，因为你必须管理线程对象的生命周期，决定适合问题和硬件的线程数量等。一个理想的方案是，你只需要将代码划分可以被并发执行的若干小片，将他们传递给编译器和运行库，然后告诉编译器“并行化这些代码来得到较优的性能”。

另外一个反复出现的场景是你可能使用几个线程来求解一个问题，但是要求它们在满足一定条件的时候提前结束。这有可能是因为结果已经被求解出来，或者因为有错误发生，甚至是因为用户显式要求操作被放弃。不管什么原因，线程需要被发送“请停止”信号，这样它们可以放弃赋予它们的任务，尽快的清理自己的环境，然后停止运行。

在本章中，我们会从自动管理线程数量和线程之间的任务划分开始介绍管理线程和任务的机制。

## 9.1 线程池

在许多公司，员工通常在办公室上班。但是有时候员工需要出差去访问客户或者供应商，参加一个交易展览或者会议。虽然这些出差是必须的，并且在某些天有好几个人同时需要出差，但是对于具体某个员工来说，不同出差的时间间隔可能是几个月甚至几年。为每个员工都配置一辆车是非常昂贵的或者不切实际的，因此公司通常都提供一个**汽车池**。汽车池有一定数量的汽车，供公司的所有员工出差使用。当公司的员工需要出差时，他们在合适的时间向汽车池申请一辆汽车。当出差回来的时候，员工将汽车归还给汽车池。如果汽车池中没有可用的汽车，员工需要重新规划自己出差的行程。

**线程池**是一个类似的思想，不过其中共享的是**线程**而不是汽车。在大多数系统上面，为每个可以与其他任务并行执行的任务分配一个单独的线程是不切实际的。但是可以尽量充分利用硬件提供的并发性。线程池允许你利用这一点。可以被并发执行的任务被提交到线程池中，在线程池中被放入一个等待队列。每个任务会被某个工作线程从等待队列中取出来执行。工作线程的任务就是当空闲的时候从等待队列中取出任务来执行。

建立一个线程池有几个关键设计问题，比如在线程池中创建几个工作线程，将任务高效分配给工作线程的方法以及是否可以等待某个任务的完成等。在这个小节中，我们会提出一些实现来解决这些问题，我们将从最简单的线程池开始。

### 9.1.1 最简单的线程池

线程池最简单的形式是含有一个固定数量的**工作线程**（典型的数量是 std::thread::hardware_concurrency()的返回值）来处理任务。当有任务要处理的时候，调用一个函数将任务放到等待队列中。每个工作线程都是从该队列中取出任务，执行完任务之后继续从等待队列取出更多的任务来处理。在最简单的情况，没有办法来等待一个任务完成。如果需要有这样的功能，则需要用户自己维护同步。

清单 9.1 给出了一个最简单的线程池的示例实现。

**清单 9.1 简单的线程池**

```
class thread_pool
{
    std::atomic_bool done;
    thread_safe_queue<std::function<void()> > work_queue;    <-1
    std::vector<std::thread> threads;    <-2
    join_threads joiner;    <-3

    void worker_thread()
    {
```

```
        while(!done)        ◄── ❹
        {
            std::function<void()> task;
            if(work_queue.try_pop(task))        ◄── ❺
            {
                task();  ◄── ❻
            }
            else
            {
                std::this_thread::yield();  ◄── ❼
            }
        }
    }
public:
    thread_pool():
        done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency();  ◄── ❽
        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this));  ◄── ❾
            }
        }
        catch(...)
        {
            done=true;  ◄── ❿
            throw;
        }
    }

    ~thread_pool()
    {
        done=true;  ◄── ⓫
    }

    template<typename FunctionType>
    void submit(FunctionType f)
    {
        work_queue.push(std::function<void()>(f));  ◄── ⓬
    }
};
```

这个实现使用一个向量来保存工作线程 ❷，使用一个第 6 章中线程安全的队列 ❶ 来管理待处理的任务。在这个实现中，用户不能等待任务，也不能返回任何值。所以你可以使用 std::function<void()>来封装你的任务。函数 submit()将任何函数或者能够调用的对象包装成为 std::function<void()>实例，然后放到队列中⓬。

工作线程是在构造器中创建的,用户使用 std::thread::hardware_concurrency() 来告诉用户硬件支持的并发线程数量❽，然后创建同样数量的线程来执行 worker_thread()

成员函数❾。

创建一个线程可能会失败，然后抛出一个异常，所以你需要保证你已经创建的任何线程都能够非常合适的停止并且清理掉。这个功能是通过一个 try-catch 块来完成的。当异常出现的时候设置 done 标志❿，在另外一边第 8 章的 join_threads❸ 的一个实例被用来等待所有工作线程的结束。这也可以在析构函数中完成，只要设置 done 标志⓫，join_threads 实例会保证在线程池被销毁前所有线程已经完成。注意成员变量声明的顺序是重要的，done 标志跟 work_queue 必须声明在 threads 向量前面，threads 向量必须声明在 joiner 前面。这个顺序保证所有成员会被正确的销毁掉。例如，在所有线程停止之前工作线程队列不能被安全的销毁。

函数 worker_thread 的实现非常简单。它包含一个循环等待直到 done 标志碑设置❹，不停的从队列中取出任务❺，然后执行它们❻。如果队列中没有任务，它会调用 std::this_thread::yield()暂停一小段时间❼，在下次执行前给其他线程一个机会往队列中添加任务。

在许多情况下，这样一个简单的线程池是足够用了，特别是如果所有任务是完全独立的并且不返回任何值或者执行一些阻塞的操作。但是存在许多简单线程池无法满足用户需求的情况，在这些情况下可能会引起一些问题，比如死锁。在简单情况下，用户像第 8 章的例子一样使用 std::async 可能会得到更好的解决办法。在本章中，我们会使用一些更加复杂的线程池实现来提供额外的特征来解决用户需要的或者减少潜在的问题。首先是等待一个我们提交的任务。

## 9.1.2 等待提交给线程池的任务

第 8 章的例子都是显式的创建线程。在划分任务给线程后，主线程总是等待创建的线程结束来保证在返回给调用者之前所有任务都已经完成了。使用线程池之后，你需要等待提交到线程池的任务结束，而不是等待工作线程。这一点跟第 8 章中基于 std::async 的例子有点相似。但是在清单 9.1 实现的线程池中，你必须人为的使用第 4 章中的条件变量等来实现这一点。这会增加代码的复杂性。如果能够直接等待任务结束会更好。

通过将复杂度移到线程池中，可以直接等待任务的结束。可以让 submit()函数返回一个任务句柄，利用这个句柄可以等待任务结束。这个任务句柄包装了条件变量或者其他来简化使用线程池的代码。

作为一个特别的例子，当主线程需要任务计算的结果的时候，主线程不得不等待任务的结束。这样的例子一直在本书中出现，比如第 2 章的 parallel_accumulate 函数。在这个例子中，你可以通过使用 std::future 来等待线程的结束，然后组合每个线程的结果。清单 9.2 展示了允许你等待任务结束和传递返回值到等待的线程需要对简单线程池做的修改。因为 std::packaged_task<>的实例只是**可移动**的，而不是**可复**

制的，不再能够用 std::function<>来作为队列中的元素，因为 std::function<>要求存储的函数对象是可以拷贝和构造的。因此，必须使用一个定制的函数包装来处理只能移动的类型。这是一个简单的带有一个调用操作符的类。因为只需要处理不带有参数并且返回 void 的函数，所以用了很直观的虚调用来实现。

**清单 9.2　有等待任务的线程池**

```
class function_wrapper
{
    struct impl_base {
        virtual void call()=0;
        virtual ~impl_base() {}
    };
    std::unique_ptr<impl_base> impl;
    template<typename F>
    struct impl_type: impl_base
    {
        F f;
        impl_type(F&& f_): f(std::move(f_)) {}
        void call() { f(); }
    };
public:
    template<typename F>
    function_wrapper(F&& f):
        impl(new impl_type<F>(std::move(f)))
    {}

    void operator()() { impl->call(); }

    function_wrapper() = default;

    function_wrapper(function_wrapper&& other):
        impl(std::move(other.impl))
    {}

    function_wrapper& operator=(function_wrapper&& other)
    {
        impl=std::move(other.impl);
        return *this;
    }

    function_wrapper(const function_wrapper&)=delete;
    function_wrapper(function_wrapper&)=delete;
    function_wrapper& operator=(const function_wrapper&)=delete;
};

class thread_pool
{
    thread_safe_queue<function_wrapper> work_queue;

    void worker_thread()
    {
        while(!done)
        {
            function_wrapper task;
```

使用函数包装器而非 std::function

```
            if(work_queue.try_pop(task))
            {
                task();
            }
            else
            {
                std::this_thread::yield();
            }
        }
    }
public:
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type>   ←❶
        submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type
            result_type;                                          ←❷
        std::packaged_task<result_type()> task(std::move(f));     ←❸
        std::future<result_type> res(task.get_future());          ←❹
        work_queue.push(std::move(task));                         ←❺
        return res;                                               ←❻
    }
    // rest as before
};
```

首先，修改后的 submit()函数 ❶ 返回一个 std::future<>对象来保存任务的返回值和允许调用者等待任务结束。这要求你知道任务函数 f 的返回值，其值为 std::result_of<>，这个值来自于 std::result_of<FunctionType()>::type，这个值是不带参数地调用 FunctionType（如 f）的返回值。通过使用 typedef 来将同样的 std::result_of<>表达式简写为 result_type❷。

然后将函数 f 包装在一个 std::packaged_task<result_type()>中 ❸，因为 f 是一个返回值类型为 result_type 并且没有参数的函数或者可以调用的对象，正如我们推导的一样。你可以在将任务加入到等待队列之前 ❺ 通过 std::packaged_task<>得到一个 future 对象 ❹，然后返回这个 future 对象 ❻。注意你必须使用 std::move()来将任务加入到等待队列中，因为 std:: packaged_task<>不是可以拷贝的。为了处理这个特性，等待队列中现在存储的是 function_wrapper 对象，而不是 std::function<void()>队列。

这个线程池的实现允许你等待你的任务结束以及得到任务的返回值。清单 9.3 展示了使用这个线程池来实现 parallel_accumulate 函数。

**清单 9.3 使用可等待任务线程池的 parallel_accumulate**

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
```

```
    if(!length)
        return init;

    unsigned long const block_size=25;
    unsigned long const num_blocks=(length+block_size-1)/block_size;    <-❶

    std::vector<std::future<T> > futures(num_blocks-1);
    thread_pool pool;

    Iterator block_start=first;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        futures[i]=pool.submit(accumulate_block<Iterator,T>());    <-❷
        block_start=block_end;
    }
    T last_result=accumulate_block<Iterator,T>()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        result+=futures[i].get();
    }
    result += last_result;
    return result;
}
```

当你将清单 9.3 的代码跟清单 8.4 的代码比较的时候，存在几个地方需要注意的。首先，你是跟数据块的数量（num_blocks❶）打交道，而不是多少个线程。为了能够充分使用线程池的扩展性，需要将任务划分为最小的值得并行的块。当线程池中只有少量线程时，每个线程需要处理许多数据块。但是当线程数量随着硬件的发展而增长时，被并发处理的数据块也增长。

你需要谨慎选择“最小的值得并行处理的块”。每个提交到等待队列中的子任务需要有一定的开销。这些开销包括提交到等待队列的开销，由工作线程去执行任务的额外开销，将结果通过 std::future<>返回给调用线程的开销。如果选择的块太小，在线程池中执行的速度可能比使用单线程的速度还要慢。

假定块的大小是合适的，你不需要担心打包任务，获得 future 或者存储 std::thread 以供后面等待线程结束使用。线程池会处理这些细节。所有你需要做的只是调用 submit()来提交你的任务❷。

线程池也处理异常的安全事宜等。任何由任务抛出的异常会被传递到 submit()返回的 future 中。如果函数带着异常退出，线程池会丢弃掉还没有完成的任务然后等待线程池中的线程结果。

这个线程池在针对每个任务都是独立的时候工作得非常好。但是当任务之间有依赖关系的时候，这个线程池就不能很好地工作了。

## 9.1.3 等待其他任务的任务

在本书中，我们一直使用快速排序算法作为例子。快速排序的原理非常简单，给定一个哨兵元素，数据被划分为两个序列，一个序列的元素都在哨兵元素之前，另外一个序列是在哨兵元素之后。然后利用递归将两个序列排序，将两个子序列的结果连接起来得到排序的最终结果。在并行化这个算法时，你需要保证这些递归调用能够充分利用可用的并行性。

回顾第 4 章，当我首次介绍这个例子的时候，我们使用 std::async 来运行其中的一个递归调用，让库来选择是使用一个新的线程来执行它还是当有关的 get() 被调用时来异步执行它。这种方式能够很好地工作，因为每个任务或者是在自己的线程上执行，或者是当需要的时候会被调用。

在第 8 章当我们重新审视快速排序的实现时，你看到一个不同的结构。在那里我们使用一组固定数目的线程。在这种情况下，我们使用一个栈来保存需要排序的序列。因为每个线程将它正在排序的数据进行划分，它将一个新的数据块放到栈上，对另外一个数据集直接进行排序。在这个做法中，直观的等待其他块结束可能会死锁，因为你在使用一个线程来等待。很容易会出现所有线程都是在等待某个数据块被排序，没有一个线程在执行实际的排序过程。我们通过让线程在等待某个未排序的数据块的时候从栈上拿出一个待处理的数据来排序来解决这个问题。

如果你将第 4 章的例子中的 std::async 替换成为本章你已经见到的简单线程池的时候，你会碰到同样的问题。线程池中只有有限数目的线程，他们可能都停在等待某个还没有被执行的任务。所以你需要一个与你在第 8 章见到的类似的解决方案，当你在等待你的数据块结束的时候去处理未完成的数据块。当你在使用线程池来管理任务列表以及它们关联的线程的时候，你不必去通过访问任务列表来完成这个。你需要做的是修改线程池的结构来自动完成这个。

最简单的访问来完成这个功能的是在线程池中增加一个新的函数来执行队列中的任务以及自己管理循环。高级线程池的实现可能会在等待函数添加逻辑来处理这种情形，有可能是通过给每个在等待的任务赋予优先级来解决。代码清单 9.4 展示了一个新的函数 run_pending_task()，代码清单 9.5 展示了使用这个函数来实现快速排序。

**清单 9.4 run_pending_task()的实现**

```
void thread_pool::run_pending_task()
{
    function_wrapper task;
    if(work_queue.try_pop(task))
    {
        task();
```

```
    }
    else
    {
        std::this_thread::yield();
    }
}
```

run_pending_stask()的这个实现是从 worker_thread()函数的主循环中提升出来的，它现在被修改为提取的 run_pending_task()。它试图从队列中取出一个任务，如果成功则执行取出的任务，否则它放弃 CPU，允许操作系统重新调度线程。清单 9.5 的快速排序的实现要比代码清单 8.1 的对应实现要简单很多，因为所有管理线程的逻辑被移动到了线程内部中。

**清单 9.5 基于线程池的快速排序的实现**

```
template<typename T>
struct sorter                                      <-❶
{
    thread_pool pool;                              <-❷

    std::list<T> do_sort(std::list<T>& chunk_data)
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }

        std::list<T> result;
        result.splice(result.begin(),chunk_data,chunk_data.begin());
        T const& partition_val=*result.begin();
        typename std::list<T>::iterator divide_point=
            std::partition(chunk_data.begin(),chunk_data.end(),
                [&](T const& val){return val<partition_val;});
        std::list<T> new_lower_chunk;
        new_lower_chunk.splice(new_lower_chunk.end(),
                               chunk_data,chunk_data.begin(),
                               divide_point);

        std::future<std::list<T> > new_lower=               <-❸
            pool.submit(std::bind(&sorter::do_sort,this,
                                  std::move(new_lower_chunk)));

        std::list<T> new_higher(do_sort(chunk_data));

        result.splice(result.end(),new_higher);
        while(!new_lower.wait_for(std::chrono::seconds(0)) ==
            std::future_status::timeout)
        {
            pool.run_pending_task();                   <-❹
        }

        result.splice(result.begin(),new_lower.get());
```

```
        return result;
    }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;

    return s.do_sort(input);
}
```

正如清单 8.1 一样，你将实际的排序工作放到了 sorter 类模板中的成员函数 do_sort()中❶，虽然在这个例子中这个类知识简单的包装了thread_pool 的实例❷。

你的线程和任务管理要做的是向线程池提交一个任务❸和执行正在等待的任务❹。这个实现比清单 8.1 简单很多。在清单 8.1 中，你不得不显式的管理线程和栈中待排序的数据块。当向线程池提交任务时，你使用 std::bind将this指针绑定给do_sort()和提供要排序的数据。在这个例子中，你使用 std::move 作用在 new_lower_chunk 作为参数传进去，来保证数据是被移动而不是新的拷贝。

虽然现在的线程池解决了任务等待其他任务中的关键死锁问题，这个线程池仍然远远不是理想的。对于使用者来说，每个 submit 的调用和每个 run_pending_task 都访问同一个队列。在第 8 章你已经见过一个集合的数据被多个线程并发的访问会大大的降低性能，所以你需要别的方法来解决这个问题。

## 9.1.4 避免工作队列上的竞争

每次线程调用 submit()时，它向单个共享工作队列添加一个新的元素。类似的情形为，工作线程不停的从队列中取出元素来执行。这意味着随着处理器数目的增加，工作队列的竞争会越来越多，这会极大地降低性能。即使你使用无锁队列，虽然没有显式的等待，但是乒乓缓存会非常耗时。

避免乒乓缓存的一个方法是在每个线程都使用一个单独的工作队列。每个线程将新的任务添加到它自己的队列中，只有当自己队列为空的时候才从全局的工作队列中取任务。清单 9.6 展示了一个使用 thread_local 变量来保证每个线程有一个自己的工作队列再加上一个全局的工作队列。

**清单 9.6 使用本地线程工作队列的线程池**

```
class thread_pool
{
    thread_safe_queue<function_wrapper> pool_work_queue;
```

```
    typedef std::queue<function_wrapper> local_queue_type;    ❶
    static thread_local std::unique_ptr<local_queue_type>
        local_work_queue;    ❷
    void worker_thread()
    {
        local_work_queue.reset(new local_queue_type);    ❸

        while(!done)
        {
            run_pending_task();
        }
    }
public:
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type>
        submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;

        std::packaged_task<result_type()> task(f);
        std::future<result_type> res(task.get_future());
        if(local_work_queue)    ❹
        {
            local_work_queue->push(std::move(task));
        }
        else
        {
            pool_work_queue.push(std::move(task));    ❺
        }
        return res;
    }
    void run_pending_task()
    {
        function_wrapper task;
        if(local_work_queue && !local_work_queue->empty())    ❻
        {
            task=std::move(local_work_queue->front());
            local_work_queue->pop();
            task();
        }
        else if(pool_work_queue.try_pop(task))    ❼
        {
            task();
        }
        else
        {
            std::this_thread::yield();
        }
    }
    // rest as before
};
```

我们使用一个 std::unique_ptr<>来保存线程私有的工作队列因为我们不想让非线程池中的线程也持有一个❷;这个是在 worker_thread()中主循环之前进行初始化❸。std::unique_ptr<>的析构函数保证了当线程退出的时候其工作队列会被适当地销毁。

submit()函数会检查当前线程是否有一个工作队列❹。如果有，则说明当前线程是一个线程池中的线程，这是可以将任务添加到私有的工作队列中，否则像之前一样将任务加入到全局工作队列中❺。

在 run_pending_task()中有一个类似的检查❻，不过这次是检查私有队列中是否有任务。如果有，可以从队列中取出一个来处理。注意到私有队列可以是普通的 std::queue<>❶ 因为私有队列只被一个线程访问。如果私有队列中没有任务，则像之前一样从全局队列取任务❼。

这能够很好地降低对全局队列的竞争，但是当任务的分布是不平衡的，可能导致一些线程的私有队列中有大量的任务而另外一些线程则没有任务处理。比如，在快速排序中，只有最顶层的任务会被添加到全局工作队列中，因为其余的数据会放在某个工作线程的私有队列中。这跟使用线程池的初衷是相反的。

幸运的是，有一些办法来解决这个问题。只要允许线程在自己私有队列以及全局队列中都没有任务时从其他线程的队列中窃取工作。

### 9.1.5 工作窃取

为了允许一个空闲的线程执行其他线程上的任务，每个工作线程的私有队列必须在 run_pending_task()中窃取任务的时候可以被访问到。这要求每个工作线程将自己的私有任务队列向线程池注册或者每个线程都会被线程池分配一个工作队列。此外，你必须保证工作队列中的数据被适当的同步和保护，这样你的不变量是被保护的。

编写一个允许拥有队列的线程在一端 push 和 pop 同时其他线程在另外一端进行任务窃取的无锁队列是有可能的。但是实现这样一个队列比较复杂，超出了本书的范围。为了验证这个想法，我们使用互斥锁来保护队列的数据。我们希望任务窃取是一个不经常发生的时间，这样互斥元的竞争就不是那么激烈，这样一个简单的队列应当只包括一个极小的额外开销。清单 9.7 所示是一个简单的基于锁的实现。

**清单 9.7 允许任务窃取的基于锁的队列**

```
class work_stealing_queue
{
private:
    typedef function_wrapper data_type;
    std::deque<data_type> the_queue;        <--❶
    mutable std::mutex the_mutex;
```

```
public:
    work_stealing_queue()
    {}

    work_stealing_queue(const work_stealing_queue& other)=delete;
    work_stealing_queue& operator=(
        const work_stealing_queue& other)=delete;

    void push(data_type data)          <--❷
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        the_queue.push_front(std::move(data));
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        return the_queue.empty();
    }

    bool try_pop(data_type& res)       <--❸
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }

        res=std::move(the_queue.front());
        the_queue.pop_front();
        return true;
    }

    bool try_steal(data_type& res)     <--❹
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }

        res=std::move(the_queue.back());
        the_queue.pop_back();
        return true;
    }
};
```

这个队列是一个对 std::deque<function_wraper>❶的封装，使用一个互斥锁来保护所有的访问。push()❷ 和 try_pop()❸ 在队列头部操作，而 try_steal()❹ 在队列尾部操作。

这实际上意味着这个队列对于拥有线程来说是一个后进先出的栈。最近被放到队列中的任务会最先被取出来执行。从缓存的角度来说这可以提高性能，因为对比之前被放入队列中的任务，被取出的任务的数据更加有可能在缓存中。同样，这个队列能够很好

的映射像快速排序之类的算法。在之前的实现中，每次调用 do_sort()将一个数据放到栈中然后等待它完成。通过处理最近放入的数据，你可以保证当前调用完成需要的数据块会比其他分支调用需要的数据块先处理好，这样可以减少活动的任务数目和总的栈空间使用量。try_steal()从队列的另外一端取数据，这样可以最小化竞争。你同样可以使用第 6 章和第 7 章中的技术来实现并发调用 try_pop()和 try_steal()。

现在你已经有一个很好的允许窃取的工作队列，怎样把它使用在你自己的线程池中呢？清单 9.8 是一个可能的实现。

**清单 9.8 使用工作窃取的线程池**

```
class thread_pool
{
    typedef function_wrapper task_type;

    std::atomic_bool done;
    thread_safe_queue<task_type> pool_work_queue;
    std::vector<std::unique_ptr<work_stealing_queue> > queues;    ←❶
    std::vector<std::thread> threads;
    join_threads joiner;

    static thread_local work_stealing_queue* local_work_queue;    ←❷
    static thread_local unsigned my_index;

    void worker_thread(unsigned my_index_)
    {
        my_index=my_index_;
        local_work_queue=queues[my_index].get();    ←❸
        while(!done)
        {
            run_pending_task();
        }
    }

    bool pop_task_from_local_queue(task_type& task)
    {
        return local_work_queue && local_work_queue->try_pop(task);
    }

    bool pop_task_from_pool_queue(task_type& task)
    {
        return pool_work_queue.try_pop(task);
    }

    bool pop_task_from_other_thread_queue(task_type& task)    ←❹
    {
        for(unsigned i=0;i<queues.size();++i)
        {
            unsigned const index=(my_index+i+1)%queues.size();    ←❺
            if(queues[index]->try_steal(task))
            {
                return true;
            }
```

```
        }
        return false;
    }
public:
    thread_pool():
        done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency();
        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                queues.push_back(std::unique_ptr<work_stealing_queue>(    ⑥
                                     new work_stealing_queue));
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this,i));
            }
        }
        catch(...)
        {
            done=true;
            throw;
        }
    }

    ~thread_pool()
    {
        done=true;
    }

    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type> submit(
        FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;
        std::packaged_task<result_type()> task(f);
        std::future<result_type> res(task.get_future());
        if(local_work_queue)
        {
            local_work_queue->push(std::move(task));
        }
        else
        {
            pool_work_queue.push(std::move(task));
        }
        return res;
    }
    void run_pending_task()
    {
        task_type task;
        if(pop_task_from_local_queue(task) ||    ⑦
```

```
            pop_task_from_pool_queue(task) ||            <-8
            pop_task_from_other_thread_queue(task))          <-9
        {
            task();
        }
        else
        {
            std::this_thread::yield();
        }
    }
};
```

这个实现跟清单 9.6 中的实现非常类似。第一个区别在于每个线程都拥有一个 work_stealing_queue，而不是普通的 std::queue<>❷。当创建每个线程的时候，不是每个线程都创建一个自己的工作队列，而是线程池的构造器为其分配一个 ❻，这个队列是存储在一个工作队列的表中 ❶。队列的下标被传递给线程函数，然后被用来获得指向队列的指针 ❸。这意味着当试图为空闲线程窃取任务时线程池可以访问该队列。run_pending_task()现在会试图从自已队列中取出任务❼，从池队列中取出任务❽，或是从其他线程的队列中取出工作❾。

pop_task_from_other_thread_queue()❹ 遍历线程池中所有线程的队列，试图一次从每个队列中窃取一个任务。为了避免每个线程都从第一个线程的队列窃取，每个线程从列表中的下一个线程窃取，通过用自己队列的下标来偏移队列下标❺。

现在你有一个线程池可以应用在许多地方。当然，仍然有许多方法针对一些特别的用法去提高它。这个是留给读者的练习。一个没有被提及的方面是当线程被阻塞了，在等待如输入输出或者互斥锁，动态地修改线程池的大小来保证有最优的 CPU 使用率。

下一个“高级”的线程管理技术是中断线程。

## 9.2 中断线程

在许多场景中，向一个长时间运行的线程发出一个信号告诉线程是停止执行的时候是一个很渴望的行为。这有可能是因为线程是一个线程池的工作线程而线程池正在被销毁，或者是因为线程正在执行的工作被用户显式地取消了，或者是因为其他一些原因。不管是何种原因，基本思想是一样的，你需要从一个线程发送一个信号告诉另外一个线程应该停止运行而不是一直执行到线程自然结束，你同样需要让线程适当的结束而不是简单的退出而造成线程池不一致的状态。

你当然可以为每种情况都设计一种单独的机制，但是这种做法没有通用性，引入许多重复工作。一种共有的机制不但可以让为后面的场景写代码变得容易，而且允许你写出能够被中断，不用担心在什么地方被使用的代码。C++11 标准病没有提供这样的机制，但是建立这样的机制是相对直观的。让我们先看看怎样可以完成这个机制，从启动和中

断一个线程的接口的角度开始。

### 9.2.1 启动和中断另一个线程

首先让我们从可中断线程的接口开始。一个可中断线程的接口需要有哪些呢？在最基本的层次上，所有你需要的是跟 std::thread 一样的接口，外加一个 interrupt() 函数。

```
class interruptible_thread
{
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f);
    void join();
    void detach();
    bool joinable() const;
    void interrupt();
};
```

在内部，你可以使用 std::thread 来管理线程本身，然后使用一些定制的数据结构来处理中断。从线程本身的角度来看中断是什么呢？在最基本的层面上你也许会说"我可以在这里被中断"，你想要一个中断点。为了让这个能够不用传递额外的参数就能使用，需要一个简单的不带任何参数的函数：interruption_point()。这意味着中断相关的数据结构需要使用一个 thread_local 的变量来访问，这个私有变量是在线程启动的时候设置好的。这样当一个线程调用你的 interruption_point() 函数的时候，它会检查当前运行的线程的数据结构。我们随后会给出 interruption_point() 的实现。

这个 thread_local 标志是你不能简单的使用 std::thread 来管理线程的主要原因；它需要使用特殊的分配方法，使得 interruptible_thread 实例可以访问，并且新启动的线程也能够访问。你可以通过在传递给 std::thread 实际启动一个线程的之前包装一下，如清单 9.9 所示。

**清单 9.9 interruptible_thread 的基本实现**

```
class interrupt_flag
{
public:
    void set();
    bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;    <-❶

class interruptible_thread
{
    std::thread internal_thread;
```

```
    interrupt_flag* flag;
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f)
    {
        std::promise<interrupt_flag*> p;          ←❷
        internal_thread=std::thread([f,&p]{           ←❸
            p.set_value(&this_thread_interrupt_flag);
            f();                                        ←❹
        });
        flag=p.get_future().get();          ←❺
    }
    void interrupt()
    {
        if(flag)
        {
            flag->set();     ←❻
        }
    }
};
```

用户给定的函数 f 被包装在一个 lambda 函数中 ❸。在此函数中，保存了一份 f 的拷贝以及一个局部 promise 的引用 p❷。lambda 函数在调用用户提供的函数之前 ❹ 为新的线程将 promise 设置为 this_thread_interrupt_flag（这个变量被声明为 thread_local❶）的地址。被调用的线程然后会等待跟 promise 相关联的 future 变得可用，然后存储在 flag 成员变量中 ❺。注意到即使 lambda 函数是运行在新线程上面，并且持有一个对局部变量 p 的应用，这是没有问题的。因为 interruptible_thread 的构造函数会一直等待直到 p 不再被新的线程引用。注意这个实现不负责处理等待线程结束或者分离线程。你需要保证当线程存在的时候或者被分离了，flag 标志被清理掉来避免危险的指针。

interrupt() 函数是一个相对直观的实现，如果你有一个合法的指针指向一个中断标志，你有一个线程可以中断，所以只要设置 flag 就可以 ❻。线程中断标志设置后，有被中断的线程来决定怎么处理这个中断。

## 9.2.2 检测一个线程是否被中断

现在你可以设置中断标志了，但是如果线程不去检查它自身是否被中断的话不会给你带来任何好处。在最简单的情况下，你可以使用 interruption_point() 函数来检查线程自身是否被中断，你可以当线程可以被安全的中断的时候调用这个函数，如果中断标志被设置的话它会抛出一个 thread_interrupted 异常。

```
void interruption_point()
{
    if(this_thread_interrupt_flag.is_set())
    {
```

```
        throw thread_interrupted();
    }
}
```

你可以在某个方便的点上调用这个函数。

```
void foo()
{
    while(!done)
    {
        interruption_point();
        process_next_item();
    }
}
```

虽然这可以工作，但是它不完美。在一些中断线程最好的地方是它被阻塞住了，正在等待某个信号。这意味着线程为了调用 interruption_point()没有在运行！在这里你需要的是一个通过使用中断的方式来等待某个事情。

## 9.2.3 中断等待条件变量

现在你可以通过在精心选定的位置上显式调用 interruption_point()来检测中断。但是当你想要一个阻塞等待的时间，如等待一个条件变量被通知，这不能给你多少帮助。你需要一个新的函数 interruptible_wait()，这个函数你可以为你想要等待的不同的事情重载。你可以知道怎样中断一个等待工作。我已经提到一个你可能想要等待的是条件变量，所以让我们从条件变量开始。为了能够中断一个条件变量的等待，你需要怎样做呢？最简单的事情是当你设置中断标志的时候通知条件变量，然后在等待的后面立即加上一个中断点。但是为了让这种方式可以工作，你不得不通知所有等待该条件变量的线程来保证你感兴趣的线程被唤醒。等待者们需要处理假的唤醒，使得其他线程能够将这个事件当成一个假的唤醒。interrupt flag 结构需要能够存储一个指向条件变量的指针这样它可以在有调用 set()的时候被通知。interruptible_wait()的一个关于条件变量的实现如下面清单 9.10 中的代码所示。

**清单 9.10 因 std::condition_variable 而遭到破坏的 interruptible_wait 函数实现**

```
void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);      ←❶
    cv.wait(lk);                                                 ←❷
    this_thread_interrupt_flag.clear_condition_variable();       ←❸
    interruption_point();
}
```

假定设置和清除一个带着中断标志的条件变量的函数存在，这个代码是优质而且简

单的。它先检查中断，然后为当前线程关联一个带 interrupt_flag 的条件变量❶，等待条件变量❷，清除关联的条件变量❸，然后再一次检查中断。如果线程是在等待条件变量的时候被中断的，调用中断的线程会广播条件变量将你唤醒，所以你可以检查中断。不幸的是，这份代码是不能工作的。它存在两个问题。第一个问题相对比较明显。因为 std::condition_variable::wait()可能会抛出异常，所以你可能没有删除中断标志和条件变量的关联就退出了。这可以通过使用一个结构的析构函数来删除关联性来修复它。

第二个问题不是那么明显。这份代码中存在一个竞争条件，如果线程是在调用 interruption_point()后面被中断，那么条件变量是否跟中断标志关联已经不要紧了，因为线程不是在等待所以不能被条件变量唤醒。你需要保证线程在上一次检查中断和调用 wait()之间不能被通知。在不改变 std::condition_variable 内部结构的情况下，你只有一种方法来做这个，使用 lk 持有的互斥锁来保持这块区域。这要求将其传递给调用 set_condition_variable。不幸的是，这又会产生一个问题，你需要传递一个生命周期未知的互斥锁给另外一个线程，那个线程在不知道它是否已经锁住互斥锁的情况下试图锁住。这有潜在的死锁可能，以及试图锁住一个已经被销毁的互斥锁。如果不能可靠地中断一个条件变量的等待，限制会非常大——你可以在没有特殊的 interruptible_wait()做得几乎一样好——那么你还有其他什么可选的方法呢？一个选项是在等待中放入一个等待的最大时间，给 wait_for()传递一个很小的时间间隔（如 1 毫秒）而不是使用 wait()。这给线程在看到中断前等待的时间设置了一个上限。如果你这样做，等待的线程会看到由定时器到期带来的大量的假唤醒，但是它不能轻易地带来帮助。清单 9.11 是这样的一个实现，还有对应的 interrupt_flag 的实现。

**清单 9.11 在为 std::condition_variable 的 interruptible_wait 中使用超时**

```
class interrupt_flag
{
    std::atomic<bool> flag;
    std::condition_variable* thread_cond;
    std::mutex set_clear_mutex;

public:
    interrupt_flag():
        thread_cond(0)
    {}

    void set()
    {
        flag.store(true,std::memory_order_relaxed);
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        if(thread_cond)
        {
```

```
            thread_cond->notify_all();
        }
    }

    bool is_set() const
    {
        return flag.load(std::memory_order_relaxed);
    }

    void set_condition_variable(std::condition_variable& cv)
    {
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        thread_cond=&cv;
    }

    void clear_condition_variable()
    {
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        thread_cond=0;
    }

    struct clear_cv_on_destruct
    {
        ~clear_cv_on_destruct()
        {
            this_thread_interrupt_flag.clear_condition_variable();
        }
    };
};

void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    interrupt_flag::clear_cv_on_destruct guard;
    interruption_point();
    cv.wait_for(lk,std::chrono::milliseconds(1));
    interruption_point();
}
```

如果你有一个要等待的断言，那么1毫秒的超时会被完全隐藏在断言的循环中：

```
template<typename Predicate>
void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk,
                        Predicate pred)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    interrupt_flag::clear_cv_on_destruct guard;
    while(!this_thread_interrupt_flag.is_set() && !pred())
    {
        cv.wait_for(lk,std::chrono::milliseconds(1));
    }
```

```
    interruption_point();
}
```

这会导致等待的条件被检查很多次，但是非常容易地用于代替 wait() 函数调用。带定时器的变形也非常容易实现，只要等待一个给定的时间，如 1 毫秒，或者其他短时间。现在 std::condition_variable 等待已久可以处理的，怎样等待一个 std::condition_variable_any 变量呢？是与此相同，或者你能做得更好？

### 9.2.4 中断在 std::condition_variable_any 上的等待

std::condition_variable_any 变量跟 std::condition_variable 不同的是它可以跟任何锁类型配合工作而不是只能跟 std::unique_lock<std::mutex>配合。结果是这会让事情变得简单，你可以更好地处理 std::condition_variable_any。因为它可以跟任何锁配合，你可以建立自己的锁类型用来加锁/解锁 interrupt_flag 的内部函数 set_clear_mutex 以及提供给等待调用的锁，如下面清单 9.12 所示。

**清单 9.12 为 std::condition_variable_any 而设的 interruptible_wait**

```
class interrupt_flag
{
    std::atomic<bool> flag;
    std::condition_variable* thread_cond;
    std::condition_variable_any* thread_cond_any;
    std::mutex set_clear_mutex;

public:
    interrupt_flag():
    thread_cond(0),thread_cond_any(0)
{}
void set()
{
    flag.store(true,std::memory_order_relaxed);
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    if(thread_cond)
    {
        thread_cond->notify_all();
    }
    else if(thread_cond_any)
    {
        thread_cond_any->notify_all();
    }
}

template<typename Lockable>
void wait(std::condition_variable_any& cv,Lockable& lk)
{
    struct custom_lock
    {
```

```
        interrupt_flag* self;
        Lockable& lk;

        custom_lock(interrupt_flag* self_,
                    std::condition_variable_any& cond,
                    Lockable& lk_):
            self(self_),lk(lk_)
        {
            self->set_clear_mutex.lock();            <-1
            self->thread_cond_any=&cond;             <-2
        }
        void unlock()                    <-3
        {
            lk.unlock();
            self->set_clear_mutex.unlock();
        }
        void lock()
        {
            std::lock(self->set_clear_mutex,lk);    <-4
        }
        ~custom_lock()
        {
            self->thread_cond_any=0;             <-5
            self->set_clear_mutex.unlock();
        }
    };
    custom_lock cl(this,cv,lk);
    interruption_point();
    cv.wait(cl);
    interruption_point();
}
    // rest as before
};

template<typename Lockable>
void interruptible_wait(std::condition_variable_any& cv,
                        Lockable& lk)
{
    this_thread_interrupt_flag.wait(cv,lk);
}
```

你的自定义锁类型在其构造函数❶中获得内部 set_clear_mutex 锁然后设置指针 thread_cond_any 指向传递给构造函数❷的 std::condition_variable_any 变量。Lockable 引用保存起来为以后使用，这必须已经被锁住。现在你可以不用担心竞争来检查中断了。如果在这个点上中断标志被设置，它是在你获得 set_clear_mutex 锁之前被设置的。当条件变量调用你的 unlock()时，你释放 Lockable 对象以及内部的 set_clear_mutex❸。此时允许试图中断你的线程在 wait()函数内部去获得 set_clear_mutex 锁和检查 thread_cond_any 指针，但是之前的却不能。一旦 wait()

函数结束等待，它会调用你的 lock() 函数，这个函数再次去获得内部的 set_clear_mutex 和 Lockable 对象的锁 ❹。现在你可以再次在你的 custom_lock 的析构函数清理 thread_cond_any 指针之前再次调用 wait() 函数去检查中断 ❺。在 custom_lock 的析构函数中你也会释放 set_clear_mutex 锁。

## 9.2.5 中断其他阻塞调用

到现在为止我们已经讨论了关于条件变量等待的情况，但是像互斥锁，future 以及其他类似的阻塞原语的等待该怎样做呢？一般情况下，你不得不使用一个用在 std::condition 变量的定时器选项，因为在不访问互斥元或者 future 的内部结构的前提下，没有办法来中断一个短时间的等待。但是使用定时器选项你知道你要等待什么，所以你可以在 interruptible_wait() 函数中循环检查。作为一个例子，下面代码是针对 std::future 的 interruptible_wait() 重载版本。

```
template<typename T>
void interruptible_wait(std::future<T>& uf)
{
   while(!this_thread_interrupt_flag.is_set())
   {
        if(uf.wait_for(lk,std::chrono::milliseconds(1)==
            std::future_status::ready)
            break;
   }
   interruption_point();
}
```

这会一直等到要么中断标志被设置，要么 future 已经准备好了，但是每次在 future 上执行阻塞等待 1ms。这意味着，假定使用高精度的时钟，在中断请求被通知之前平均每次大概需要等待 0.5ms。wait_for 函数经常会等待一整个时钟滴答，所以如果你的时钟滴答的间隔是 15ms，你会每次至少等待 15ms 而不是 1ms。取决于应用场景，这有可能会变得无可坚守。你总是可以降低定时器到期的时间间隔。但是其坏处是线程被唤醒更多次来检查中断标志，这会增加线程切换的额外开销。

到现在为止我们已经讨论了你应该怎样通过使用 interruption_point() 和 interruptible_wait() 函数检测中断，但是你应该怎样处理中断呢？

## 9.2.6 处理中断

从被中断的线程的角度来看，一个中断只是一个 thread_interrupted 异常。这可以像其他异常一样进行处理。典型的操作是你可以使用一个标准的 catch 块来捕获它。

```
try
{
    do_something();
}
catch(thread_interrupted&)
{
    handle_interruption();
}
```

这意味着你可以捕获中断，用某些方法来处理它，然后继续执行。如果你这样做，另外一个线程再次调用 interrupt()，你的线程在调用一个中断点的时候会再次被中断。你可能想这样做如果你的线程是在执行一系列独立的任务的话。中断一个任务会导致那个任务被放弃，线程会继续执行列表中的下一个任务。

因为 thread_interrupted 是一个异常，当调用能产生中断的代码段的时候所有异常安全的预防措施必须被执行来保证资源没有被泄露以及数据结构保持在一个一致的状态。经常发生的一种情况是让中断来结束线程，这种情况下你只要让异常抛出就可以了。但是如果你让异常传播的范围超过了 std::thread 的构造器的话，std::terminate 将会被调用，整个程序都会被终止。为了避免被迫记得在每个你传递到 interruptible_thread 的函数中放一个 catch(thread_interrupted) 句柄，作为替代，你可以将此 catch 块放到你用来初始化 interrupt_flag 的包装器中。这样做会让允许中断异常未被处理而传播变得安全，因为它接下来仅仅会终止单独一条线程。在 interruptible_thread 构造函数中的线程初始化现在看起来是这个样子。

```
internal_thread=std::thread([f,&p]{
        p.set_value(&this_thread_interrupt_flag);
    try
    {
        f();
    }
    catch(thread_interrupted const&)
    {}
});
```

现在让我们看一个中断很有用的完整例子。

### 9.2.7 在应用退出时中断后台任务

现在考虑桌面搜索应用。此应用也与用户进行互动，它需要监视文件系统的状态，识别所有的改变并且更新它的索引。为了避免影响 GUI 的响应性，这种处理就留给基础线程来完成。这个基础线程需要在应用的生命期始终运行，在应用初始化的时候就启用它，然后一直运行直到应用结束。对于这样一个应用通常只有在机器被关机的时候才会发生，因为此应用需要始终运行来保持最新的索引。在任何情况下，当应用结束的时候，就需要按顺序关闭基础线程，实现它的一种方法就是中断它。

清单 9.13 展示了这样一个系统的线程管理部分的一个简单实现。

**清单 9.13 在后台监视文件系统**

```
std::mutex config_mutex;
std::vector<interruptible_thread> background_threads;

void background_thread(int disk_id)
{
    while(true)
    {
        interruption_point();                    <-❶
        fs_change fsc=get_fs_changes(disk_id);   <-❷
        if(fsc.has_changes())
        {
            update_index(fsc);   <-❸
        }
    }
}

void start_background_processing()
{
    background_threads.push_back(
        interruptible_thread(background_thread,disk_1));
    background_threads.push_back(
        interruptible_thread(background_thread,disk_2));
}

int main()
{
    start_background_processing();   <-❹
    process_gui_until_exit();        <-❺
    std::unique_lock<std::mutex> lk(config_mutex);
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].interrupt();   <-❻
    }
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].join();   <-❼
    }
}
```

启动的时候，开始运行基础线程 ❹。然后主线程将基础线程与处理 GUI 一起处理 ❺。当用户要求应用退出的时候，中断这些基础程序 ❻，然后主线程等待每个基础线程在退出前完成❼。基础线程在一个循环里聚集，检查磁盘变化❷并且更新索引 ❸。每次循环它们通过调用 interruption_point()检查中断 ❶。

为什么你在等待前要中断所有线程？为什么不逐个中断然后再移动到下一个前进行等待？答案就是并发性。当线程被中断，它们不会立即结束，因为它们在退出前必须前进到下一个中断点然后运行析构函数调用和异常处理代码。通过立即联合所有线程，

你就可以使中断线程等待，即使它仍然可以做它能做的有用的工作——中断别的线程。你等待直到不再有任何工作的时候（所有线程都被中断），这才允许所有线程被中断来并行地处理它们的中断并且更快结束。

这种中断的方法可以简单扩展为增加进一步中断调用或者通过一个具体代码块来禁止中断，但是这将留待读者考虑。

## 9.3 总结

本章，我们考虑了许多“高级的”线程管理方法：线程池和中断线程。你已经看到使用本地工作队列如何减少同步管理以及潜在提高线程池的吞吐量，并且看到当等待子任务完成时如何运行队列中别的任务来减少发生死锁的可能性。

我们也考虑了许多方法来允许一个线程中断另一个线程的处理，例如使用特殊中断点和如何将原本会被中断阻塞的函数变得可以被中断。

# 第 10 章　多线程应用的测试与调试

本章主要内容

■ 并发相关的错误
■ 通过调试和审阅代码来定位错误
■ 设计多线程的测试
■ 测试多线程代码的性能

到目前为止，我主要介绍了写并行代码有哪些可用到的工具，怎么使用它们，以及代码的整体设计和结构。本章将要介绍软件开发的另一个关键步骤：测试和调试。如果你想通过学习本章来寻找测试并行代码的一个简单方法的话，那么，要让你失望了。测试和调试并行代码是非常难的。本章主要向你介绍一些比较常用而且重要的测试和调试技巧。

测试和调试就相当于一个硬币的两面——测试代码寻找错误，调试代码纠正错误。幸运的话，你自己调试出所有的错误，而不是让使用该应用的人发现代码漏洞。在我们介绍测试和调试之前，重要的是理解可能会出现哪些问题，让我们先来看看这些问题。

## 10.1　并发相关错误的类型

在并发代码中，你几乎会碰到任何类型的错误，但是，有些类型的错误仅会在并发代码中出现，本书仅关心这些与并发相关的错误。这些并发相关的错误主要分为两大类。

- 不必要的阻塞。
- 竞争条件。

这两大类又分为很多小类，首先我们来看不必要的阻塞。

### 10.1.1 不必要的阻塞

不必要的阻塞是什么意思？首先，线性阻塞是指线程因为要等待某些条件（如互斥元、条件变量、时间等）无法继续运行时所处的状态。多线程代码中，常用这些条件，而这些条件常常无法获得满足，因此就出现了不必要的阻塞问题。我们接着又会提出下一个问题：为什么这个阻塞是不必要的？因为有其他一些线程在等待该阻塞的线程执行一些动作，如果该线程阻塞的话，其他线程也势必阻塞。不必要的阻塞又分成以下几种。

- **死锁**——如第 3 章所说的，死锁是指第一个线程在等待第二个线程执行后才能继续，而第二个线程又在等待第一个线程，如此构成一个线程等待循环状态。如果你的线程死锁了，那你的程序将无法继续执行下去。在许多可以预见的情况下，多线程中的某一线程是负责与用户接口交互的，在死锁情况下，用户接口会停止应答。而在其他情况下，用户接口仍会应答，只不过有些必要的任务无法得到执行，如不会返回搜索结果或者不会打印文件等。
- **活锁**——当第一个线程等待第二个线程时，而这第二个线程又在等第一个线程情况时，活锁类似于死锁。活锁与死锁的关键不同在于等待过程不是一个阻塞状态而是一个不断的循环检测状态，如自旋锁。严重时，活锁的症状就像死锁（应用不会执行任何进程），不同仅在于 CPU 此时的利用率非常的高，因为现在还在不断的运行检测，只因相互等待而阻塞。不太严重时，当某个随机事件发生时，活锁可能会被解锁，但是，活锁会导致任务较长时间得不到执行，并且在这期间 CPU 利用率高。
- **在 I/O 或外部输入上的阻塞**——当你的线程阻塞是因为等待某外部输入而无法继续执行，可能这个外部输入永远都不到来，那么这种阻塞就称之为基于等待 I/O 或其他外部输入的阻塞。因此，不希望出现一个线程因等待外部输入而阻塞，其他线程有因为要等待这个线程的运行而阻塞的情况出现。

上面简要的介绍了几种不必要的阻塞类型，那么什么是竞争条件呢？

### 10.1.2 竞争条件

竞争条件是多线程代码中的问题最常见的原因——许多死锁和活锁实际上是竞争条件的表现。并不是所有的竞争条件都是有问题的—竞争条件发生的时间取决于各个独

立线程操作的先后顺序。许多竞争条件是有益的。例如，到底哪个线程来处理任务队列中的下一个任务是不确定的。然而，许多并发错误的产生是由于竞争条件。竞争条件常常产生下面几种错误类型。

- **数据竞争**——数据竞争是一种特殊的竞争条件。因为没有同步好对某个共享内存的并行访问，因此，数据竞争会造成未定义的操作出现。在第5章我们学习C++内存模型时，我介绍过数据竞争。当错误地使用原子操作来同步线程或者想通过共享数据来避免互斥元死锁时，常常会发生数据竞争。
- **破坏不变量**——常常表现为悬挂指针（因为另一个线程删除了被访问的数据）、随机存储损坏（线程由于局部更新而造成的读取数据不一致）或者双闲状态（如当两个线程从同一队列中弹出相同的值，并且这两个线程因此而删除一些相关数据）等。破坏不变量常指不变量在时间或数值上的改变。如果多个线程要求以特定的顺序执行，那么不正确同步可能会产生由于线程执行顺序错误而引起的竞争条件。
- **生存期问题**——人们常常会将生存期问题归结为破坏不变量问题，但实际上生存期问题是竞争条件产生的另一个独立的问题分类。在这个分类中的错误的基本问题是线程会超时访问某些数据，而这些数据可能已经被删除、销毁或者访问的内存其实已经被另一个对象重用。当一个线程要参考某一局部变量，而这个局部变量已经不在该线程访问能力之内了，这样就会造成生存期问题。当线程的生存时间与它可以操作的数据之间没有某种限制规则时，那么，就极有可能出现在线程结束之前该数据就已被销毁，而造成线程访问错误的问题。如果在线程中调用 `join()` 来让数据等到线程完成后再销毁，那你需要保证当发生异常时，可以跳过 `join()` 函数的执行，这是线程异常的基本安全保障。

竞争条件是问题杀手。死锁和活锁会导致任务长时间得不到执行。通常，你可以添加一个调试器来运行区分哪些线程陷入死锁或者活锁，并且哪些并发对象是相互矛盾的。

在整个代码的任何地方都可能出现上面介绍的数据竞争、破坏变量和生命周期的问题的症状（如随机崩溃或者不正确的输出），代码可能会重写后面其他程序可能会用到的内存，导致编译出错。编译给出的错误定位往往完全与出错代码无关，可在程序执行很久后，才能暴露该错误。这类错误往往是由共享系统内存造成的，就算你小心翼翼地试图指定某线程访问某数据，并且保证正确同步，但是，任何线程都有可能重写应用程序中其他线程需要使用的数据。

至此，我们简要明确了我们将要遇到的错误类型，下面让我们看看，我们该怎样来定位错误实例，并解决它们。

## 10.2 定位并发相关的错误的技巧

在前面的内容中，我们学习了在代码中可能会遇到的并发相关的错误类型，以及这些错误的表现形式。对上述知识有所了解后，你可以检查你的代码，并找出错误可能出现在哪里，你可以先尝试确定某段代码是否有错。

也许最显然最直接的方法，就是查看代码。这虽然看似明显，但实际上是很难贯彻的。当你阅读自己刚写的代码时，很容易读成你想要写的，而非你真正写的。相似地，如果要你阅读他人写的代码时，你快速阅读可能定位和解决一些简单的问题，一些重大或比较隐晦的问题，则需要我们花大量的时间去梳理代码，考虑可能出现的并行问题和非并行问题。在下面的代码中，我们将具体问题具体对待。

就算是检阅你自己的代码，你还是可能会漏掉一些错误。因此，无论何时，你都要确保你的代码可以执行，即使代码无法顺利执行，你也要保持平和的心态。因此，我们会介绍一下与检阅代码相关的一些多线程测试和调试技巧。

### 10.2.1 审阅代码以定位潜在的错误

正如前面提到的，当检阅多线程代码来纠正并行相关的错误时，彻底仔细地阅读非常重要，要像一把细齿梳子一样仔细地阅读代码。如果可能让他人帮你检阅你的代码，因为他们没有参与代码的编写，他们不得不想清楚代码是如何工作的，因此，会发现很多遗漏的错误。这需要代码的阅读者有充足的时间来仔细负责地检阅代码，而不是简单快速地过一遍。大多数并行错误不是简单快速的扫视代码所能发现的，这些错误往往需要微妙的时机才会出现。

如果你让你的同事帮你检阅你的代码，这个代码对他来说是完全陌生的。因此，他们会从不同的视角来看问题，并指出一些你未发现的错误。如果你找不到同事帮你检阅代码，你可以找朋友帮忙，甚至将代码发到网络上寻求帮助。如果你实在找不到人帮你检阅代码，或者，他们也无法找出问题，别急，你还可以这么做。对于初学者来说，将代码搁置一段时间，去做其他事情，如编写该程序的其他部分、读书、散步等。在这段时间内，当你集中精神做其他事时，你的潜意识还在想着这个问题。同时，当你重新回到该代码时，代码已经不那么熟悉了，这样你可能就会以一种不同的视角来检阅你的代码。

让别人审阅代码的替代方法是自己审阅。一个有用的技巧是试图解释它是如何工作的细节给别人。这个别人甚至可以不是实体的人，如布偶熊或橡胶鸡，我个人认为编写详细的注释极有帮助。你要解释，每一行代码有什么作用，会发生什么，访问的数据等。你要不断地自我提问并解释回答。通过不断地问自己这些问题，并仔细思考它的答案，

问题常常自己就会暴露出来，你会发现这真是一个难以置信的发现错误的有效方法。这些问题对于检阅任何代码都是有用的，而不仅对于检阅你自己的代码。

### 审阅多线程代码时需要思考的问题

正如我说的，代码阅读者在阅读代码时思考一些与代码相关的特定问题是非常有用的。这些问题会使阅读者集中注意力到一些代码相关的细节上，并且帮助发现一些潜在的错误。下面列出一些具体的而非全部的，我喜欢问的一些问题。你也可以找到其他一些你比较关注的问题。不再多说了，先将这些问题列出以便参考。

- 哪些数据是需要保护，防止并行访问的？
- 如何保证你的数据是被保护的？
- 此时其他线程执行到代码的何处？
- 该线程用的是哪些信号量？
- 其他线程持有哪些信号量？
- 该线程各操作之间有先后顺序的要求吗？在其他线程中存在这样的问题吗？这些要求如何强制执行？
- 该线程载入的数据是否有效?该数据是否已经被其他线程修改了？
- 如果你假设其他线程可能正在修改该数据，那么可能会导致什么样的后果以及如何保证这样的事情永不发生？

最后一个问题是我最喜欢问的问题，因为它确实能帮我理清楚线程之间的关系。通过假设某行代码存在错误，你就可以像个侦探一样追查原因。为了说服你自己，代码没有错误，你需要考虑到所有情况和可能排序。当数据在其生命期内受多个信号量保护时，这个方法非常有用，例如，使用第 6 章中给出对线程安全序列，这个安全的队列的头和尾对应不同的信号量，你必须要保证线程持有的另一个信号量不会访问相同的队列元素。这个问题还会使得对公有数据或者其他代码能够很容易得到该数据指针或引用的私有数据进行特定审查的问题更加重要和明确。

列举的倒数第二个问题同样也很重要，因为它解决了一个常常会犯的简单错误，如果你释放后再重新获取该信号量你必须假设其他线程已经修改了该共享数据。很明显，如果互斥锁因为它们对于对象来说是内部的不是立即可见的—你可能在不知不觉中就那么做了。在第 6 章中，可以看到当函数提供线程安全数据结构时太细粒度的时候，是如何导致竞争条件以及错误的。但是，对于一个非线程安全的栈来说，栈可能会被多个线程并行访问，让 top() 和 pop() 操作独立开来是必要的，那么，共享数据被修改的情况将不再会出现，因为内部互斥元的锁在这两个调用之间就已经被释放了，因此，另一个线程就可以修改栈了。第 6 章中，解决的办法是将两个操作结合起来，所以它们都在同一互斥锁的保护下执行，从而消除了潜在的竞争条件。

那么，让我们来回顾一下你自己的代码（或者他人的代码），你要确保没有代码错

误。你该怎样测试你的代码确保无错或否定你代码无错的信念，只有试过才知道。

### 10.2.2 通过测试定位并发相关的错误

开发单线程应用时，应用测试比较简单耗时。首先，你需要区分所有可能的输入数据集（至少包括一些典型的输入测试集）并且对这些输入数据集进行测试。如果应用程序能够正确执行并且产生正确的输出，说明这个应用程序对于给定的输入集能够正常运行。如果测试到错误状态，处理则会比正确运行的情况复杂。但是，基本思想是相同的——建立初始化条件执行应用程序。

测试多线程代码相对于单线程来说难得多，因为合理的调度线程是不确定的，因此线程调度的差异会导致运行的变化。因此，即使应用程序运行同一组输入数据，如果代码中潜伏有竞争条件的话，仍然有可能会导致有时运行正确有时运行出错。因为有潜在的竞争条件并不意味着代码执行总是失败，仅仅是**有时有可能会失败**。

鉴于固有的难以再现并发相关的错误，因此，需要仔细地设计测试程序。你希望每次测试能够确定问题可能存在的最少的代码，那么当测试失败时，你就可以更好地隔离出错代码——测试并行队列最好能够直接测试并行压栈和出栈工作而不是测试使用并行队列的整个代码块。这样有助你思考该怎样设计测试代码——参考本章后面的易测性设计小节中的内容。

我们值得通过测试消除并发来证明问题是并发相关的。如果你让所有程序运行在一个线程时出错，该错只是一个普通的错误而非一个并发相关的错误。追踪错误的初始发生位置而不是被你的测试工具测试发现的错误位置是非常重要的。这是因为即使错误发生在你应用的多线程部分，也并不意味着它就是并发相关的。如果你使用线程池来管理并发等级，通常你可以通过设置配置参数来指定工作线程。如果你手动地管理线程，你就需要修改代码以便使用单个线程测试来进行测试。一方面，你可以将你的线程减少到一个，这样就可以根除并发；另一方面，如果在**单核**系统中没有错误（即使是一个多线程应用），但是在**多核**系统或**多处理器**系统中出错，那么就是竞争条件错误和可能同步或内存顺序错误。

比代码结构更加重要的是测试代码的并发性，测试代码的结构仅仅跟测试环境一样重要。如果你连续用一个测试实例来测试并发队列，你需要考虑以下各种不同的应用场景。

- 一个线程在自身队列上调用 push() 或 pop() 来验证该队列工作在基础级别。
- 在一个空队列上一个线程调用 push() 同时另一个线程调用 pop()。
- 在一个空队列上多个线程调用 push()。
- 在一个满队列上多个线程调用 push()。
- 在一个空队列上多个线程调用 pop()。

- 在一个满队列上多个线程调用 pop()。
- 在一个特定的满队列上多个线程调用 pop()，该队列的总长度不够，无法满足所有线程。
- 在一个空队列上同时有多个线程调用 push() 和一个线程调用 pop()。
- 在一个满队列上同时有多个线程调用 push() 和一个线程调用 pop()。
- 在一个空队列上同时有多个线程调用 push() 和多个线程调用 pop()。
- 在一个满队列上同时有多个线程调用 push() 和多个线程调用 pop()。

考虑完上述所有场景或者更多场景后，接着你需要考虑关于测试环境的附加因子。

- 在每个场景中，多线程是什么意思（3、4、1024？）。
- 系统是否有足够的处理核来为每个运行线程分配一个核。
- 测试程序需要在哪种结构的处理器上运行。
- 你将怎样为你测试的并行部分确定合适的时间顺序。

对于特殊情况需要考虑附加因子。鉴于对以上四种环境的考虑，第一种和最后一种环境会影响测试代码自身的结构（参见 10.2.5 节），其他两种与正在使用的物理测试系统有关。使用到的线程数与特定的被测试代码有关，但是，可以通过构建测试代码的不同的方式来得到合理的时间顺序表。在我们学习这些技术之前，让我们来看看怎样设计一个便于测试的应用代码。

### 10.2.3 可测试性设计

测试多线程代码是困难的，所以你会想怎样才能使代码易于测试呢？你能做的最重要的事情之一就是设计易于测试的代码。现有设计易于测试代码的技术大都用于单线程代码，但是，其中许多技术也同样可以应用多线程。通常，做到以下几点后，代码就比较易于测试了。

- 每个函数功能和类的划分清晰明确。
- 函数扼要简洁。
- 你的测试代码可以完全控制你的被测试代码的周围的环境。
- 被测试的需要特定操作的代码应该集中在一块而不是分散在整个系统中。
- 在你写测试代码之前你要先考虑如何测试代码。

所有以上提到的都可以应用在多线程代码中。事实上，我认为上述几点更多的应用于解决多线程代码的易测性而非单线程代码的易测性。上述最后一条非常重要，即使你编写应用代码之前，此时还远没有到写测试代码的那一步，在你编写应用代码之前也有必要考虑怎样测试它——使用什么样的输入，哪些条件下可能会出错，怎样找到代码潜在的错误等。

设计易于测试的并行代码最好的方法之一就是消除并发。如果你可以将代码分割成

多个部分，在一个单线程内由这些部分来负责要操作的通信数据与多个线程之间的通信路径，这样，你就极大地减少了问题。操作被一个单线程访问的数据时的这些应用部分可以使用正常的单线程技术来进行测试。这样，那些难以测试的用于处理线程之间通信和确保一个时间内仅有一个线程访问特定数据块的并发代码部分就变得比较少，测试出现错误时，也更加容易进行追踪错误源头。

例如，如果你的应用被设计成一个多线程的的状态机，那么你就可以将它分解成多个部分。用于为每个可能的输入集确保状态转换和操作的正确性的线程的状态逻辑可以通过单线程技术独立的进行测试，并且通过测试工具提供的测试输入集，可以同样应用到其他线程。接着，通过测试代码中特别设计多并发线程和简单的状态逻辑，核心状态机和确保各事件按正确的顺序到达正确的线程的信息路由的代码可以独立的进行测试。

可选地，如果你将代码分解成多个代码块，**读共享数据/迁移数据/更新共享数据**，你可以使用所有的单线程技术来测试**迁移数据**代码块部分，因为此时这部分代码仅是一个单线程代码。测试一个多线程迁移困难的问题可以降级为测试读共享数据块和更新共享数据块中的一个，哪个简单选哪个。

需要注意的是库函数调用能够使用内部变量来存储状态，然后，如果多个线程使用相同的库函数调用集在多线程之间实现共享。因为代码访问共享数据不是立即表现出来的，因此，多线程的共享还存在一些问题。然而，随着你对这些库函数调用的学习，多线程共享仍然是个问题。这时，你要么添加适当的保护和同步或者使用可替代的对于多线程的并行访问来说安全函数。

设计多线程的易测性比你构建代码以减少用来处理并发相关的问题代码和注意对于一些非线程安全的库函数调用代码的代码量来说更为重要。在浏览代码时，记得问一下自己 10.2.1 小节中的问题是非常有用的。尽管这些问题可能不是直接关于测试或易测性的，但是，如果你事先在你的测试代码中考虑到上述问题并且考虑如何测试你的代码，那么，做出的设计选择可能不同以使测试更加简单。

既然我们学习了合理的设计代码可以使测试变得更加容易，潜在地修改代码来从“单线程部分”（这个单线程仍可以通过并发模块与其他线程进行交互）隔离“并发部分”（比如线程安全容器或状态机事件逻辑），下面让我们来学习测试并发代码的相关技术。

### 10.2.4 多线程测试技术

你需要思考你想要测试的场景并且编写一些小的代码来测试函数功能。那么，你怎样确保那些存在潜在的问题的时间调度通过小的测试练习解决它的潜在错误呢？

事实上，有许多方法可以做到这点，如暴力测试或者压力测试。

### 1. 暴力测试（brute-force testing）

暴力测试的核心思想是穷举所有可能情况看代码是否能够正常而不出现错误。最典型的方法是多次运行代码，并且尽可能地一次运行多个线程。如果一个错误仅在多个线程以某一特定顺序运行时出现，那么运行的代码越多，出错的可能性就越大。如果你仅测试一次并且通过了测试，你可能自信地以为代码没有问题，能够工作。如果你一批运行十次并且每次都能通过测试，你就会更加自信。如果你测试了十亿次，并且每次都通过测试，那你就会对你的代码自信无比。

你的自信程度取决于你通过测试的次数。如果你的测试结果非常精确，测试甚至可以精确地概括到线程安全队列的话，这样的穷举测试会让你对自己的代码无比自信；另一方面，如果被测试的代码非常的多，可能的排列数非常多，运行即使十亿次也仅会产生一点点自信。

**穷举测试的缺点是它可能会让人产生盲目的自信。**可能你编写的测试环境不会产生错误，就算你运行多次也不会出现错误，但是，换一个稍微不同的环境就会每次测试都出错。最坏的情况就是在你的测试系统中不会出现有问题的测试环境因为你测试是在一个特殊的环境。除非你的代码运行的环境与你代码测试运行的环境一模一样，并且相应的硬件和操作系统也不会引起任何错误出现。

这里给出的一个典型的例子就是在一个单处理系统上测试一个多线程应用。因为每个线程都要求运行在同一个处理器上，所有的任务都是自动串行进行的，那么在多处理器上可能遇到的许多竞争条件和双向缓存问题在单处理器系统中都不复存在了。这不仅仅是变量的问题；不同的处理器体系结构产生不同的同步和设备时序问题。例如，在 x86 和 x86-64 体系结构上，自动加载的操作通常是一样的，但是是否标识 memory_order_relaxed 或者 memory_order_seq_cst 是不同的（参见 5.3.3 节）。这意味着那些编写的代码可以在放松内存顺序的 x86 系统上正确运行，而在有着精确时序操作指令集系统如 SPARC 系统中会运行失败。

如果你需要你的应用能够方便的在多个目标系统运行，那么在这多种系统上进行一些有代表性实例的测试是非常重要的。这就是我为什么在 10.2.2 节测试环境中列出被使用的处理器体系结构的原因。

避免潜在的盲目自信的关键是成功地进行穷举测试。这需要仔细考虑测试设计，不仅考虑与被测代码单元的选择，还要考虑测试工具的设计和选择测试环境。你需要保证尽可能多的方法测试代码，也要尽可能考虑所有可行的线程交互。

尽管穷举测试确实能给你带来自信，但是，穷举测试无法保证找到所有问题。这里介绍一种可以找到所有问题的技术，我们称之为**组合仿真测试**。这种测试技术要求你花时间将它应用到你的代码和合适的软件中去。

## 2．组合仿真测试

这有点绕嘴，因此我最好先解释一下我的意思。组合仿真测试是指在一种特殊的仿真代码真实运行环境的软件上运行你的代码。你将注意到这个软件允许你在一个单物理计算机上运行多个虚拟机，这些虚拟机和硬件的特性是被上层软件竞争调用。不同于仿真系统，模拟软件能够记录线程数据访问、锁、原子操作等的先后顺序。然后，使用C++内存模型的规则重复运行每组允许的组合操作来识别竞争条件和死锁。

虽然如此全面的测试组合能够保证找到系统中的所有错误，但是，许多小的错误，往往需要花费大量的时间来发现它，因为组合操作的排列数会随着线程数和每个线程的操作数增长而呈现指数增长的趋势。因此组合测试技术最好保留到对代码片段进行精细测试时再用，而不是应用对整个应用程序的测试。组合测试的一个明显的缺点就是它需要依赖于仿真软件处理你代码中操作的能力。

组合测试技术可以用来在正常条件下反复测试你的代码，但是，这种技术可能会漏查一些错误，因此，你需要一种技术，这种技术可以让你在各种特定的条件下反复测试你的代码。有这样一种技术存在吗？

使用在测试运行时发现问题的库函数就是这样一种技术。

## 3．使用特殊的库函数来检测测试暴露出的问题

尽管这种技术无法提供全面检查组合的模拟测试，但是，你可以使用一些特别的库函数同步基本单元来找到大部分错误，这些同步基本单元如互斥元、锁和条件变量等。例如，常用的要求对一块共享数据使用互斥锁。当你访问数据时，如果检测到互斥锁，就可以证实当访问数据时，调用线程已将该互斥元锁住了并且报告访问失败。通过标记你的共享数据，你可以使用库函数来检查数据共享。

如果有一个特殊线程一次拥有多个互斥元，应用库函数还可以记录锁的顺序。如果另一个线程在不同的时序锁住该互斥元，即使测试运行时没有出错，也会将之标记成一个可能的死锁。

测试多线程的另一类特殊的库函数是通过多个线程中将获得锁的那个线程或者通过 notify_one()函数调用一个竞态变量的线程的控制权交给测试人员来实现线程的原子属性，如互斥元和条件变量。这样可以让你建立特定的测试场景并且验证代码在这些特定场景内是否能顺利运行。

此外，在C++标准库函数中也有一部分可用于测试的库函数，我们可以在我们的测试工具中调用这些标准库函数。

看完执行测试代码的不同方式之后，现在我们来看看构建测试代码来实现你希望的调度顺序的方法。

### 10.2.5 构建多线程的测试代码

前面的 10.2.2 节中，我告诉大家你要找到方式来为你测试程序的“while”部分提供某种可行的调度顺序，下面我们将要学习在这一过程会遇到的问题。

基本的问题是，你需要安排一组线程，这组线程中的每个线程在你指定的时间内都可以执行一段选定的代码。最常见的情况是，你有两个线程，但是这可以很容易地扩展到更多。在第一步中，你需要区分每个测试的不同的部分。

- 通用的启动代码需要在所有代码之前启动的那段代码。
- 线程特定的启动代码必须在每个线程上启动的那段代码。
- 每个线程的实际代码是指你希望并行运行的那段代码。
- 在并行执行完成后运行的那段代码，包括代码状态的断言。

为了更进一步解释，我们来看看 10.2.2 节的测试列表执行一个特定的例子，一个线程用于对一个空队列调用 `push()`，另一个线程则用于调用 `pop()`。

通用启动代码很简单，就是你必须创建队列。执行 `pop()` 的线程没有线程特定的启动代码。而对于执行 `push()` 函数的线程来说，它的线程特定的启动代码依赖于队列的接口和存储对象的类型。如果将要存储的对象很难构建或者必须是堆分配的，那么，你可以将这个存储对象的构建过程或堆分配过程作为线程特定的启动代码，这样存储对象的构建过程或堆分配过程就不会影响你的测试了。相反，队列仅存储普通的 `int` 类型，那么就不需要在启动代码中构建 `int` 型。被测试的实际代码是相当明确的——就是对 `push()` 和 `pop()` 的调用。那么，在这个例子中，哪个是“结束后”的代码部分呢？

在这个例子中，那段“结束后”的代码部分就取决于你希望用 `pop()` 函数来做什么。如果你用它来阻塞线程直到队列有数据为止，那么，你可以明确“结束后”代码获取向 `push()` 函数提供的返回数值和队列置空。如果 `pop()` 不用于阻塞线程，并且在队列为空时结束，那么你需要测试两种可能性，要么 `pop()` 函数返回向 `push()` 函数提供的数据，要么队列为空或者 `pop()` 函数指示没有数据并且队列中有一个元素。当其中的任意一种可能性为真时，你希望避免的是场景是 `pop()` 函数显示“没有数据”而且队列是空的，或者 `pop()` 函数返回值，但是，队列却仍然不为空。为了简化测试，假设你有一个阻塞 `pop()` 函数。那么最后的代码就是出队列的数据即为进队列的数据，并且队列为空。

至此，我们已经区分了代码的不同部分，接着你就要尽量让一切代码都按计划运行。一个可行的办法是使用一系列的 `std::promises` 来指示一切就绪。每个线程设置一个 promise 来指示该线程已准备就绪，接着等待从第三方 `std::promise` 获得的一个（或者一个副本）`std::shared_future`；主线程等待所有线程的所有 promise 被设置，然后控制这些线程运行。这就保证了在并行程序运行之前每个线程都已被启动，任意线程特定的启动代码必须在线程的 promise 设置之前就被执行。最后，主线程要等待所有

线程结束并检查最终的状态。你同时还需要注意线程异常还确保不会有任意一个线程需要等待还未发生的操作信号。清单 10.1 给出了这个例子的测试代码。

**清单 10.1 队列上当前调用的 push()和 pop()的测试例子**

```
void test_concurrent_push_and_pop_on_empty_queue()
{
    threadsafe_queue<int> q;                                   ❶

    std::promise<void> go,push_ready,pop_ready;                ❷
    std::shared_future<void> ready(go.get_future());           ❸

    std::future<void> push_done;                               ❹
    std::future<int> pop_done;

    try
    {
        push_done=std::async(std::launch::async,               ❺
                             [&q,ready,&push_ready]()
                             {
                                 push_ready.set_value();
                                 ready.wait();
                                 q.push(42);
                             }
            );
        pop_done=std::async(std::launch::async,                ❻
                            [&q,ready,&pop_ready]()
                            {
                                pop_ready.set_value();
                                ready.wait();
                                return q.pop();                ❼
                            }
            );
        push_ready.get_future().wait();                        ❽
        pop_ready.get_future().wait();
        go.set_value();                                        ❾

        push_done.get();                                       ❿
        assert(pop_done.get()==42);                            ⓫
        assert(q.empty());
    }
    catch(...)
    {
        go.set_value();                                        ⓬
        throw;
    }
}
```

这一结构很好地呼应了我们前面的介绍。首先，创建空队列，这部分作为通用启动代码 ❶。然后，为所有"就绪"信号创建各自的 promise❷，并且为 go 信号获取一个 std::shared_future❸。接下来，你可以创建 future 来表示线程已经运行结束 ❹。这些需要程序跳转到 try 模块之外，这样你就可以为异常设置 go 信号而无需等待测试

线程运行结束（因为在测试代码可能出现死锁—将死锁限制在测试代码内部是一种相当理想的情况）。

在 try 模块内部你可以启动线程❺、❻——你可以使用 std::lanch::async 来保证任务在其各自的线程上运行。注意使用 std::async 可以你的异常安全任务相比于使用普通的 std::thread 来说更为简单，这是因为 future 的析构函数在整个线程执行过程中都会加入该线程。Lambda 捕获详细说明每个任务都会参考队列和相关的 promise 已就绪信号，并且将从 go promise 中复制 ready future。

如上所述，每个任务设置它自己的 ready 信号，然后，在运行时间测试代码之前等待通用 ready 信号。主程序的过程与之相反—在设置信号来启动真正的测试❽之前等待来自两个线程的信号❾。

最终，主线程从异步调用中去调用 future 上的 get() 来等待任务的完成❿、⓫和检查结果。注意 **pop** 任务通过 future 来返回检索值❼，因此，你可以使用它来获取断言的结果⓫。如果抛出异常，你设置 go 信号来避免任何产生悬挂线程的和再次抛出异常的机会⓬。任务对应的 future❹ 在后面声明，那么，首先会销毁这些 future 并且它们的回收器会在任务未就绪时等待任务完成。

虽然这似乎是相当多的样板只是为了测试两个简单的调用，有必要使用一些类似的测试以实现在最好的测试时机测试你真正想要的部分。例如，实际的线程启动可能是一个非常耗时的过程，因此，如果你不让线程等待 go 信号，那么，push 线程就会在 pop 线程启动之前就已经完成了，这样就完全错过了测试时机。使用 future 确保两个线程在相同的 future 上运行和阻塞。解除 future 阻塞允许两个线程同时运行。一旦你对该结构熟悉后，你很容易就可以在该模式上直接创造出新的测试代码。该模式也可以很容易地扩展到多个线程的测试。

至此，我们已经学习了多线程代码的**正确性**。尽管多线程代码的正确性是一个很重要的问题，但它不是你进行测试的唯一理由。测试多线程代码的**性能**也同样重要，这部分我们将会在下节介绍。

### 10.2.6　测试多线程代码的性能

在应用程序中使用并行的一个主要原因是为了充分利用现在流行的多核处理器来提高应用程序的性能。因此，实际测试你的代码确认性能确实得到提升，就像你对应用程序尝试了其他性能优化一样。

使用并行提高性能将会带来一个特殊的**扩展性**问题——你可能希望在 24 核机器上代码运行速度是在单核机器上的 24 倍，24 个核是平等的。你不希望代码运行在 24 核上的速度仅仅是双核机器上的两倍。回顾 8.4.2 小节，如果你代码中重要部分代码仅在一个线程上运行，会限制代码潜在的性能收益。因此，有必要在你开始测试前查看你代码

的整体设计，你会知道你是否能够获得 24 倍的性能提升，或者你代码的一系列整体设计和架构限制你的代码仅能获得 3 倍的性能。

就像你在前面章节所看到的，进程之间竞争访问的数据会极大地影响性能。然而，当处理器的数目少时，可能系统性能较好，而当处理器数目较多时，系统性能反而很差，因为处理器的数目多了，竞争也就多了。

因此，当测试多线程代码的性能时，最好先检测多种不同配置下的系统性能，由此你能评估出系统性能扩展能力。至少，你该测试下单处理器系统和多处理器系统下的性能。

## 10.3 总结

在本章，我们学习了各种你可能会碰到的与并行相关的错误类型，如死锁活锁、数据竞争和其他问题的竞争条件等。接着，我们又介绍了定位这些错误的一些技巧。这些技巧包括：代码检阅过程中不断地自我提问及思考解答、指导撰写测试代码，以及如何为并行代码构建测试代码。最后，我们学习了一些有助于测试的通用部件。

# 附录 A　C++11 部分语言特性简明参考

C++新标准增加的并不只是对并发的支持，除此之外还有一整套的语言特性以及新的类库。在这一附录中，我会对一些 C++11 新语言特性进行一番概述，这些特性有助于我们理解 Thread 库以及本书的其他内容。它们之中除了 `thread_local`（参见 A.8 节）外，与并发都没有直接的联系，但在进行多线程编程的时候却很实用。这里涉及的内容，都是对简化代码或提高代码可读性所必需（如右值引用）或者很重要的。使用了这些特性的代码或许刚开始会因为不为大家所熟知，而显得很难懂，但当你熟悉它们之后，结果就会反过来。随着 C++11 逐渐流行开来，利用这些特性的代码就会变得稀松平常。

话不多说，让我们首先来看看**右值引用**（**rvalue references**），在线程库中，为了更好地在对象间进行线程、锁或者其他任何东西的所有权转换，右值引用被广泛地使用。

## A.1　右值引用

如果你曾做过 C++编程，就会对引用很熟悉。C++的引用允许我们为一个现有的对象创建一个新的名字，所有通过引用完成的访问和修改操作都会影响其本体。例如，

```
int var=42;
int& ref=var;        // 创建一个到变量的引用
ref=99;
assert(var==99);     // 因为对引用进行赋值，本体也被修改了
```

迄今为止我们所使用的引用都是**左值引用**（**lvalue references**）。**左值**（**lvalue**）这一术语来源于 C 语言，用来指代那些可以用在赋值表达式左侧的东西，具名对象、在栈和堆上分配的对象，或者其他对象的成员，总之就是有确定存储空间的东西。而术语**右值**（**rvalue**）也是源自 C 语言，指的是只能在赋值表达式右侧出现的东西，如字面值和临时对象。左值引用只能被绑定到左值，不能绑定到右值。例如，你不能这样写：

```
int& i=42;          <— 不能编译
```

因为 42 是一个右值。好吧，这不是太准确；你一直能够将一个右值绑定到 `const` 左值引用上：

```
int const& i=42;
```

但是，在右值引用之前，为了能够将临时对象作为引用参数传递给函数，C++标准故意设置了一个例外。允许进行隐式转换，所以你可以这样写：

```
void print(std::string const& s);      创建临时的 std::string
print("hello");                        对象
```

无论如何，C++11 标准引入了**只能**绑定到右值，而不能绑定到左值的**右值**引用（**rvalue references**），在声明的时候从使用一个&符号改为使用两个&符号：

```
int&& i=42;
int j=42;          不能编译
int&& k=j;
```

于是，你可以通过函数重载，让一个重载版本接受左值引用，另一个接受右值引用，来决定函数的形参是左值还是右值，这就是**移动语义**（**move semantics**）的基础。

## A.1.1　移动语义

右值通常是临时对象，因此可以被自由地修改。如果已知函数的形参是一个右值，那么就可以将它用作临时存储，或者“窃取”其内容而不影响程序的正确性。这就意味着，你可以**移动**（**move**）右值参数的内容，而不是**复制**（**copy**）其内容。对于大型的动态结构，这样做可以节约大量的内存开支，并且能够提供很大的优化空间。考虑一个函数，它接受一个 `std::vector<int>`类型的形参，并且在内部复制一份以便于在不影响原数据的情况下进行修改。以往的做法是，将这个参数作为一个常量左值引用，并且在内部进行复制。

```
void process_copy(std::vector<int> const& vec_)
{
    std::vector<int> vec(vec_);
    vec.push_back(42);
}
```

这就允许函数同时接受左值和右值，但每次都被迫进行一次复制。如果你用一个接

受右值引用的版本重载该函数，你就可以避免在右值的情况下做复制，因为你明白可以自由地修改原始值，

```
void process_copy(std::vector<int> && vec)
{
    vec.push_back(42);
}
```

现在，如果该函数是类的构造函数，你就可以窃取右值的内容，并且在新实例中使用它们。考虑清单 A.1 列出的类，在默认构造函数中分配了一大块内存，它会在析构函数中被释放。

**清单 A.1　具有移动构造函数的类**

```
class X
{
private:
    int* data;
public:
    X():
        data(new int[1000000])
    {}
    ~X()
    {
        delete [] data;
    }
    X(const X& other):          <-❶
        data(new int[1000000])
    {
        std::copy(other.data,other.data+1000000,data);
    }
    X(X&& other):          <-❷
        data(other.data)
    {
        other.data=nullptr;
    }
};
```

拷贝构造函数（**copy constructor**）❶正是按照你所期望的那样定义的，分配一个新的内存块，然后将数据复制进去。然而，你还可以编写一个通过右值引用接受原值的构造函数 ❷，这就是移动构造函数（**move constructor**）。在这个例子里，仅仅是把数据的指针复制一份，然后赋以 other 实例一个空指针，这样就可以节约大量的内存空间和从右值创建变量的时间。

对于类 X 而言，移动构造函数仅仅是一项优化，但在有些场合，即便当提供一个拷贝构造函数是毫无意义的时候，移动构造函数也有其意义。例如，std::unique_ptr<> 的全部意义就在于每个非空实例都是指向其对象的唯一指针，因此拷贝构造函数是没有意义的。然而，移动构造函数允许在实例间传送指针的所有权，并且允许

std::unique_ptr<>被用作函数的返回值——指针被移动而不是被复制了。

如果你希望显式地从一个你确信不会再使用的命名对象中移动数据，你可以通过使用 static_cast<X&&>或者调用 std::move()来将其转换为右值：

```
X x1;
X x2=std::move(x1);
X x3=static_cast<X&&>(x2);
```

当你希望将参数值移入局部变量或成员变量的时候，就可以从中获益，因为一个右值引用参数虽然可以绑定到右值，但在函数内部，却是被视为左值的：

```
void do_stuff(X&& x_)
{
    X a(x_);                    <-- 复制
    X b(std::move(x_));                  <-- 复制
}                                                    正确，右值绑定到右
do_stuff(X());                                   <-- 值引用
X x;                      错误，左值不能绑定到
do_stuff(x);          <-- 右值引用
```

移动语义在 Thread 类库中被广泛使用，既可以用在对于复制没有语义上的意义但资源可以被转移的地方，也可以作为一项优化，以避免反正会被销毁掉的源所带来的复制开销。在 2.2 节的一个例子中，你曾看到我们用 std::move()将一个 std::unique_ptr<>实例传送到一个新建的线程中，然后在 2.3 节中，我们又看到了在 std::thread 实例间传送线程的所有权。

std::thread、std::unique_lock<>、std::future<>、std::promise<>和 std::packaged_task<>都是不可复制的，但它们都具有移动构造函数，允许相关资源在实例间进行传输，并且支持它们作为函数的返回值。std::string 和 std::vector<>可以被复制，但它们同样具有移动构造函数和移动赋值操作符，以此来避免大量数据作为右值时的复制开销。

C++标准库并不会对显式移入另一个对象的对象做任何处理，除了销毁和对其赋值（复制或者移动，后者更常见）之外。然而，确保一个处于移入状态的类的不变性，是好的习惯。例如，一个用作移动来源的 std::thread 实例等效于一个默认构造的 std::thread 实例，一个用作移动来源的 std::string 实例仍然具备有效状态，尽管无法保证究竟那个状态是什么（即不知道该字符串有多长或者包含什么字符）。

### A.1.2　右值引用与函数模板

使用右值引用作为函数模板的参数的最终差别在于如果函数形参是对模板参数的右值引用，如果提供了一个左值，自动模板参数类型推断会将类型推断为左值引用，如果提供的是右值，则推断为普通的无修饰类型。这听起来有些绕口，所以我们来看一个示例。考虑下面的这个函数：

```
template<typename T>
void foo(T&& t)
{}
```

如果按照如下所示用一个右值进行调用，那么T就会推断为该值的类型：

```
foo(42);                  <-- 调用 foo<int>(42)
foo(3.14159);             <-- 调用 foo<double>(3.14159)
foo(std::string());       <-- 调用 foo<std::string>(std::string())
```

然而，如果你用一个左值来调用 foo，T 就会被推断为一个左值引用：

```
int i=42;
foo(i);                   <-- 调用 foo<int&>(i)
```

因为函数参数声明为 T&&，也就是一个引用的引用，它被视为原始的引用类型。于是 foo<int&>()的签名就是：

```
void foo<int&>(int& t);
```

这就允许单个函数模板同时接受左值和右值参数，并且被用作 std::thread 的构造函数（参见 2.1 节和 2.2 节），以便于当参数是右值的时候，受支持的可调用对象能够被移动到内部存储中，而非复制。

## A.2 deleted 函数

有时候，允许一个函数被复制是没有意义的。std:mutex 就是个典型的例子——如果你真的对互斥元进行复制这意味着什么？std::unique_lock<>是另一个例子，某个实例是其所持有锁的唯一拥有者。如果真的对其进行复制，就意味着那个副本也控制该锁，这是没有意义的。在实例间转移所有权，正如 A.1.2 节中提到的，是有意义的，但那并不是复制。我可以肯定你还遇到过其他的例子。

过去阻止一个类被复制的惯常方法是将拷贝构造函数和拷贝赋值操作符声明为私有的，并且不提供其实现。如果有任何类外部的代码试图复制一个实例，将会导致一个编译时错误，如果任何类成员函数或者友元试图复制一个实例，则会导致一个连接时错误（缺少实现所导致）：

```
class no_copies
{
public:
    no_copies(){}
private:
    no_copies(no_copies const&);                    ❶ 没有实现
    no_copies& operator=(no_copies const&);
};
no_copies a;                                        ❷ 不会编译
no_copies b(a);
```

在 C++11 中，委员会认识到这虽然是惯常做法，但同时也认识到这有一些不优雅。于是，委员会提供了一个更加通用的机制，你还可以将其应用到其他场合，你可以通过在函数声明前添加=deleted，将一个函数声明为已删除的（**deleted**）。于是 no_copies 可以写为：

```
class no_copies
{
public:
    no_copies(){}
    no_copies(no_copies const&) = delete;
    no_copies& operator=(no_copies const&) = delete;
};
```

这就比原来的代码更加具有描述性，并且清楚地表达了意图。这也允许编译器给出更具描述性的错误信息，并且将你在类的成员函数内执行拷贝的错误从连接时转移到了编译时。

如果在删除了拷贝构造函数和拷贝赋值操作符的同时，显式编写了移动构造函数和移动赋值操作符，该类就变成只能移动的，就像 std::thread 和 std::unique_lock<>一样。清单 A.2 展示了这种只移动类型的实例。

**清单 A.2　简单的只移动类型**

```
class move_only
{
    std::unique_ptr<my_class> data;
public:
    move_only(const move_only&) = delete;
    move_only(move_only&& other):
        data(std::move(other.data))
    {}
    move_only& operator=(const move_only&) = delete;
    move_only& operator=(move_only&& other)
    {
        data=std::move(other.data);
        return *this;
    }
};
move_only m1;
move_only m2(m1);                  <-- 错误，拷贝构造函数声明为被删除的
move_only m3(std::move(m1));       <-- 正确，找到移动构造函数
```

只移动对象可以作为函数参数传入，也可以从函数中返回。但如果你希望从一个左值移入，你必须总是显式地使用 std::move()或者 static_cast<T&&>。

你可以将=delete 标识符应用到任意函数，而不仅仅是拷贝构造函数和拷贝赋值操作符。这可以清楚地表示该函数是不可用的，但并不仅限于此。一个被删除的函数按照通常的方式参与重载方案，并且当它被选中时仅导致一个编译时错误，这可以用来移

除特定的重载。例如，如果你的函数接受一个 short 类型参数，你可以通过编写一个接受 int 类型变量的重载版本并将其声明为被删除的，来阻止截断 int 型值：

```
void foo(short);
void foo(int) = delete;
```

但凡尝试用 int 来调用 foo，现在都会遇到编译错误，调用者必须显式地将值转换为 short：

```
foo(42);              // 错误，int 重载声明为被删除的
foo((short)42);       // 正确
```

## A.3 defaulted 函数

删除的函数允许你显式声明一个函数未被实现，而默认的（**defaulted**）函数则恰恰相反，它们允许你告诉编译器必须为你编写这个函数，作为其“默认”实现。当然，你只能对编译器可以自动生成的函数这样做，包括默认构造函数、析构函数、拷贝构造函数、移动构造函数、拷贝赋值操作符和移动赋值操作符。

那你为什么会要这么做呢？以下是一些可能的原因。

- 为了改变函数的可访问性。在默认情况下，编译器生成的函数是 public 的。如果你希望将它们变为 protected 或者 private，你就得亲自去编写它们。通过将它们声明为 defaulted，你就可以让编译器去编写这些函数，同时又改变它们的访问级别。
- 作为注解。即使编译器生成的版本足够使用，仍然值得像这样将其显式进行声明，以便于你或者其他人将来再看代码的时候能够清楚地了解这是有意为之。
- 为了强制编译器去生成该函数，否则它们可能不会这么做。典型的是针对默认构造函数，只有在没有用户自定义构造函数的时候它才通常会由编译器生成。如果你需要自定义一个拷贝构造函数（举个例子），你仍然可以通过声明一个 defaulted 的默认构造函数，而让编译器生成它。
- 为了将一个析构函数设为虚拟的，将其留给编译器来生成。
- 强制声明一个特定的拷贝构造函数，如令其接受一个非 const 引用的源参数而不是一个 const 引用。
- 利用编译器生成函数的一些特定属性，如果你自己提供实现的话，可能会失去它们——这一点我们稍后会提及。

类似于 deleted 函数通过后缀=delete 来进行声明，defaulted 函数只需要在声明后添加=default 来进行声明，如：

```
class Y
{
```

```
private:
    Y() = default;                        <-- 改变访问级别
public:
    Y(Y&) = default;                      <-- 接受一个非常量引用
    T& operator=(const Y&) = default;     <-- 声明为 defaulted 作为注解
protected:
    virtual ~Y() = default;               <-- 改变访问级别并且添加 virtual
};
```

我之前提到过编译器生成的函数可以具有一些特殊的属性，而在用户定义的版本中却无法获得。其中最大的区别就是编译器生成的函数可以是平凡的（**trivial**）。这会带来包括下面所述的一些后果。

- 具备平凡的拷贝构造函数、平凡的拷贝赋值操作符和平凡的析构函数的对象可以被 memcpy 和 memmove 复制。
- constexpr 函数（参见 A.4 节）所使用的字面值类型必须具有一个平凡的构造函数、拷贝构造函数和析构函数。
- 拥有平凡的默认构造函数、拷贝构造函数、拷贝赋值操作符和析构函数的类可以用在一个具有用户定义构造函数和析构函数的联合体中。
- 具有平凡的拷贝赋值操作符的类可以在 std::atomic<>类模板（参见 5.2.6 节）中使用，用来提供该类型值的原子操作。

仅将函数声明为=default 并不能使其成为平凡的（只有在类同时支持所有其他条件的时候，相应的函数才能成为平凡的），但如果在用户代码中显式地编写该函数，则会阻止其成为平凡的。

具有编译器生成函数和用户提供的等效函数的类之间的第二个不同，就是没有用户提供的构造函数的类可以作为**聚合体**（**aggregate**），并且可以通过一个聚合初始化器进行初始化：

```
struct aggregate
{
    aggregate() = default;
    aggregate(aggregate const&) = default;

    int a;
    double b;
};
aggregate x={42,3.141};
```

在这里，x.a 被初始化为 42，x.b 被初始化为 3.141。

编译器生成的函数和用户提供的等效函数之间的第三点区别非常深奥，仅仅适用于默认构造函数，而且仅仅是那些满足特定条件的类的默认构造函数。考虑下面这个类：

```
struct X
{
    int a;
};
```

如果你不使用初始化器创建类 X 的实例，那么其中的 int(a) 会被默认构造（**default initialized**）。如果该对象拥有静态存储期，那么它会被初始化为零；否则，它会拥有一个不确定的值，如果在其未被赋予新值之前访问它，可能会导致未定义的行为：

```
X x1;                                   // x1.a 具有不确定的值
```

另一方面，如果你通过显式地调用默认构造函数来初始化 X 的实例，那么 a 会被初始化为零：

```
X x2=X();                               // x2.a == 0
```

这种怪异的特性同样延伸至基类和成员。如果你的类拥有编译器生成的默认构造函数，并且所有数据成员和基类同样拥有编译器生成的默认构造函数，则那些基类和内置类型成员的数据成员也会如此，或是留下一个不确定的值，或是初始化为零，这取决于外部的类的默认构造函数是否被显式调用。

尽管这条规则很混乱并且容易导致错误，但它却有其作用，并且如果你自己编写默认构造函数，你会失去这条特性。像 a 这样的数据成员总是被初始化（因为你指定了一个值或者显式默认构造函数）或者总是未被初始化（因为你没有这么做）：

```
X::X():a(){}        // 总是 a == 0
X::X():a(42){}      // 总是 a == 42
X::X(){}            // ❶
```

如果你像第三个例子❶中那样，从 X 的构造函数中省略 a 的初始化，那么对于 X 的非静态实例，a 未被初始化，对于具有静态存储期的 X，a 被初始化为零。

在通常情况下，如果你手动编写任何其他的构造函数，编译器就不再为你生成默认构造函数，所以如果你需要的话，就得自己去编写它，这意味着你将失去这一奇异的初始化特性。然而，通过显式地将该构造函数声明为 defaulted，可以令编译器强制地为你生成默认构造函数，这一特性也会保留下来：

```
X::X() = default;                 // 为 a 应用默认初始化规则
```

这一特性被用于原子类型（参见 5.2 节），它们的构造函数就显式地为 defaulted。它们的初始值总是未定义的，除非(a)它们具有静态存储时间段（因此被静态地初始化为零）；(b)显式地调用默认构造函数，请求零初始化；(c)显式地指定一个值。注意在原子类型的情况下，用一个值进行初始化的构造函数被声明为 constexpr（参见 A.4 节），以便允许静态初始化。

## A.4 constexpr 函数

像 42 这样的整型字面值是常量表达式（**constant expressions**）。简单的算术表达式，

如 23×2-4，也是如此。你甚至可以使用整数类型的 const 变量，它们又是通过由常量表达式组成的新的常量表达式来进行初始化的：

```
const int i=23;
const int two_i=i*2;
const int four=4;
const int forty_two=two_i-four;
```

除了使用常量表达式来创建能用于其他常量表达式的变量，还有些事情你只能通过常量表达式来完成。

- 指定数组的边界：

```
int bounds=99;
int array[bounds];          // 错误，bounds 不是一个常量表达式
const int bounds2=99;
int array2[bounds2];        // 正确，bounds2 是一个常量表达式
```

- 指定非类型模板参数的值：

```
template<unsigned size>
struct test
{};
test<bounds> ia;            // 错误， bounds 不是一个常量表达式
test<bounds2> ia2;          // 正确，bounds2 是一个常量表达式
```

- 在类定义中，为一个 static const 的整数类型类数据成员提供一个初始化器：

```
class X
{
    static const int the_answer=forty_two;
};
```

- 为内置类型或者集合提供一个初始化器，它可被用于静态初始化：

```
struct my_aggregate
{
    int a;
    int b;
};
static my_aggregate ma1={forty_two,123};    // 静态初始化
int dummy=257;
static my_aggregate ma2={dummy,dummy};      // 动态初始化
```

- 像这样的静态初始化可以用来避免初始化顺序的问题以及竞争条件。

上述的这些都不是新生事物，在 1998 版本的 C++标准中你都可以做到。然而，在新标准中，通过引入 constexpr 关键字，常量表达式（**constant expression**）的构成被扩展了。

constexpr 关键字主要作为函数修饰符。如果函数的参数和返回值满足特定的要求，并且函数体足够简洁，该函数就可以声明为 constexpr，这样它就可以在常量表

达式中被使用，比如：

```
constexpr int square(int x)
{
    return x*x;
}
int array[square(5)];
```

在这里，array 会拥有 25 条记录，因为 square 被声明为 constexpr。当然，仅仅因为该函数可以被用于常量表达式并不意味着它的所有用法都自动地成为常量表达式。

```
int dummy=4;
int array[square(dummy)];      <-- ❶
```

❶ 错误，dummy 不是一个常量表达式

在这个例子中，dummy 不是一个常量表达式 ❶，所以 square(dummy) 也不是（仅是一个常规的函数调用），因此也不可被用来指定 array 的边界。

## A.4.1　constexpr 与用户定义类型

到目前为止，所有的示例都是用的内置类型比如 int。然而，新 C++标准允许常量表达式可以是任何满足**字面值类型**要求的类型。如果要一个类型具有字面值类型资质，以下几点必须全部满足。

- 必须具备平凡的拷贝构造函数。
- 必须具备平凡的析构函数。
- 所有非静态数据成员和基类必须是平凡的类型。
- 必须具有一个平凡的默认构造函数或者 constexpr 构造函数，而非拷贝构造函数。

我们将马上看一看 constexpr 构造函数。现在，我们关注一下具有平凡的默认构造函数的类，比如下面列出的 CX 类。

**清单 A.3　具有平凡默认构造函数的类**

```
class CX
{
private:
    int a;
    int b;
public:
    CX() = default;          <--❶
    CX(int a_, int b_):        <--❷
        a(a_),b(b_)
    {}
    int get_a() const
    {
```

```
        return a;
    }
    int get_b() const
    {
        return b;
    }
    int foo() const
    {
        return a+b;
    }
};
```

注意，我们显式地将默认构造函数 ❶ 声明为 **defaulted**（参见 A.3 节），这是为了在面对用户定义构造函数 ❷ 的时候保持它的平凡性质。因此该类型满足作为字面值类型的全部条件，你就可以将其用在常量表达式中，比如，提供一个 constexpr 函数来创建新的实例：

```
constexpr CX create_cx()
{
    return CX();
}
```

你还可以创建一个简单的 constexpr 函数来复制其参数：

```
constexpr CX clone(CX val)
{
    return val;
}
```

不过这差不多就是所有你可以做的事情了——一个 constexpr 函数仅能调用其他的 constexpr 函数。所以你能够做的，就是将 constexpr 应用至 CX 的成员函数和构造函数：

```
class CX
{
private:
    int a;
    int b;
public:
    CX() = default;
    constexpr CX(int a_,int b_):
        a(a_),b(b_)
    {}
    constexpr int get_a() const          <-❶
    {
        return a;
    }
    constexpr int get_b()                <-❷
    {
        return b;
    }
    constexpr int foo()
    {
```

```
    return a+b;
  }
};
```

注意，get_a() ❶ 后的 const 限定符现在是多余的，因为使用 constexpr 已经暗示这一点了。即便省略了 const 限定符，get_b() 仍然是 const 的。现在可以定义如下所示的更加复杂的 constexpr 函数。

```
constexpr CX make_cx(int a)
{
    return CX(a,1);
}
constexpr CX half_double(CX old)
{
    return CX(old.get_a()/2,old.get_b()*2);
}
constexpr int foo_squared(CX val)
{
    return square(val.foo());
}
int array[foo_squared(half_double(make_cx(10)))];    <-- 49个元素
```

然而有趣的是，如果你所需要的仅仅是一个更加优雅的计算数组界限的方式或者一个完整的常数，这样做会花费大量的精力。引入用户定义类型的常量表达式和 constexpr 函数的主要优点，就是由常量表达式初始化的字面值类型对象会被静态初始化，于是它们的初始化就避免了竞争条件和初始化顺序问题。

```
CX si=half_double(CX(42,19));    <-- 静态初始化
```

这同样包括了构造函数。如果构造函数被声明为 constexpr，且构造函数的参数是常量表达式，那么该初始化就是一个常量初始化（**constant initialization**）并且作为静态初始化阶段的一部分来进行。**这是 C++11 中为并发而设的最重要的变化之一**，通过允许用户自定义构造函数仍可进行静态初始化，你可以在其初始化时避免所有的竞争条件，因为保证了它们在所有代码运行前就被初始化。

以上对于像 std::mutex（参见 3.2.1 节）或 std::atomic<>（参见 5.2.6 节）之类的特别重要——你可能想要使用一个全局实例来同步访问其他变量，并在访问过程中避免竞争条件，而如果互斥元受限于竞争条件，前面所述的就将无法实现，所以 std::mutex 的默认构造函数声明为 constexpr，以确保互斥元的初始化总是作为静态实例化阶段的一部分来完成的。

## A.4.2　constexpr 对象

到目前为止我们看到 constexpr 应用在函数上，它同样可以被应用于对象。这主要用于检测目的，它可以确认该对象是通过常量表达式、constexpr 构造函数或由常

量表达式构成的初始化器来初始化的。它也会将对象声明为 const：

```
constexpr int i=45;                          <- 正确
constexpr std::string s("hello");            <- 错误，std::string 不是字面值类型
int foo();
constexpr int j=foo();                       <- 错误，foo()并未声明为 constexpr
```

## A.4.3 constexpr 函数要求

为了将一个函数声明为 constexpr，它必须满足一些要求。如果不满足这些要求，将其声明为 constexpr 会引起编译时错误。constexpr 函数所需的要求如下所列。

- 所有的参数必须是字面值类型。
- 返回值类型必须是字面值类型。
- 函数体只能由单个 return 语句组成。
- return 语句中的表达式必须是常量表达式。
- 所有的构造函数和用于构造表达式返回值的转换运算符必须是 constexpr。

这些看起来很直接：你必须能够将函数内嵌到常量表达式中，仍然形成一个常量表达式，你也不能够修改任何东西。constexpr 函数是无副作用的纯函数。

对于 constexpr 类成员函数，还有些附加的要求。

- constexpr 成员函数不能是虚拟的。
- 函数作为成员的这个类必须是字面值类型。

对于 constexpr 构造函数，规则又有点不同。

- 构造函数体必须是空的。
- 每个基类必须被初始化。
- 每个非静态的数据成员必须被初始化。
- 成员初始化列表中使用的所有表达式必须具有常量表达式资质。
- 被选中用以初始化数据成员和基类的构造函数必须是 constexpr 构造函数。
- 构造数据成员和基类的表达式里所使用到的所有构造函数和转换运算符都必须是 constexpr 的。

这是与针对函数而言相同的一组规则，区别在于没有返回值，所以没有 return 语句。与之不同的是，构造函数在成员初始化列表中初始化所有的基类和数据成员。平凡的拷贝构造函数是隐式的 constexpr。

## A.4.4 constexpr 与模板

如果模板的某个特定实例化的参数和返回值不是字面值类型，当 constexpr 被应用到函数模板或者类模板的成员函数上的时候，此关键字会被忽略。这就允许你编写这

样的函数模板，当模板参数类型合适的时候它是 constexpr，否则就是普通的 inline 函数，例如：

```
template<typename T>
constexpr T sum(T a,T b)
{
    return a+b;
}
constexpr int i=sum(3,42);        // 正确，sum<int>是 constexpr
std::string s=
    sum(std::string("hello"),
        std::string(" world"));   // 正确，但 sum<std::string>不是 constexpr
```

该函数必须满足作为 constexpr 函数的其他所有要求。你不能声明一个具有多条语句的函数为 constexpr 仅仅因为它是一个函数模板，这仍然是编译错误。

## A.5 lambda 函数

lambda 函数是 C++11 标准中最让人激动的特性之一，因为它们可以极大地简化代码，以及大量消除与编写可调用对象相关的样板。C++11 lambda 函数语法允许一个函数在另一个表达式中需要它的地方进行定义。这对于有些东西非常有用，如提供给等待函数 std::condition variable 的断言（参见 4.1.1 节），因为它允许语义以可访问的变量的形式快速被表达，而不是通过一个函数调用操作来获取类中成员变量的所需状态。

一个最简单的 **lambda** 表达式定义了一个不接受参数的、只依赖于全局变量和函数的自包含函数，甚至不必返回值。这样的 lambda 表达式就是包括在一对花括号中的一系列语句，之前缀以方括号（**lambda** 引导符）。

```
[]{                      // 以[]开始 lambda 表达式
    do_stuff();
    do_more_stuff();
}();                     // 结束 lambda 表达式，并调用之
```

在这个例子中，lambda 表达式被紧随其后的一对括号调用了，但通常并不这样做。首先，如果你打算直接调用它，你一般用不着 lambda，而是将语句直接写在源代码处。更通常的情况是，将其作为参数，传给接受可调用对象作为参数的函数模板。在这种情况下，它可能需要接受参数，或者返回一个值，或两者兼有。如果需要接受参数，你可以像普通函数那样，在 lambda 引导符后面加上参数列表。例如，下面一段代码将向量的所有元素写入 std::cout，以换行为分隔。

```
std::vector<int> data=make_data();
std::for_each(data.begin(),data.end(),[](int i){std::cout<<i<<"\n";});
```

返回值几乎同样简单。如果 lambda 函数体由单条 return 语句组成，那么 lambda

的返回类型就是待返回表达式的类型。例如，你可能会用类似下面所示的简单 lambda，来等待 std::condition_variable（参见 4.1.1 节）要设置的标志位。

**清单 A.4 具有推断返回类型的简单 lambda**

```
std::condition_variable cond;
bool data_ready;
std::mutex m;

void wait_for_data()
{
    std::unique_lock<std::mutex> lk(m);
    cond.wait(lk,[]{return data_ready;});    <-❶
}
```

传给 cond.wait()的 lambda 返回类型是通过 data_ready 的类型推断的，即 bool。一旦条件变量从等待中被唤醒，接下来就会调用该互斥元锁定的 lambda，并且只会在 data_ready 为 true 的时候，从对 wait()的调用中返回。

那要是你无法将 lambda 语句体写成单条的 return 语句呢？这种情况下你就得显式指定返回类型。在语句体只有一条 return 语句的时候，你依然可以这样做，但如果 lambda 语句体更为复杂的时候，你必须这样做。返回类型是通过在 lambda 参数列表后跟以一个箭头（->）加返回类型的方式进行指定的。如果你的 lambda 不接受任何参数，你仍然需要包含空的参数列表，以便显式指定返回值。你的条件变量预测就可以写成，

```
cond.wait(lk,[]()->bool{return data_ready;});
```

通过指定返回类型，你可以扩展这个 lambda，来记录消息或者做些更复杂的处理。

```
cond.wait(lk,[]()->bool{
    if(data_ready)
    {
        std::cout<<"Data ready"<<std::endl;
        return true;
    }
    else
    {
        std::cout<<"Data not ready, resuming wait"<<std::endl;
        return false;
    }
});
```

尽管像这样的简单 lambda 都很强大并能够极大地精简代码，可它们的真正实力是在捕捉局部变量的时候才能发挥出来。

## 引用局部变量的 lambda 函数

使用[]作为 lambda 引导符的 lambda 函数不能引用任何包含它的作用域内的局部变

量，它们只能使用全局变量以及作为参数传入的东西。如果你希望访问某个局部变量，你需要捕捉它。最简单的做法就是通过使用[=]作为 lambda 引导符，来捕捉局部作用域中的整个变量集。这就是所有你需要做的——你的 lambda 现在可以在其被创建的时候访问局部变量的副本。

为了实际地了解这一点，考虑下面这个简单的函数。

```
std::function<int(int)> make_offseter(int offset)
{
    return [=](int j){return offset+j;};
}
```

每次调用 make_offseter 都会通过 std::function<>函数包装器来返回一个新的 lambda 函数对象。返回的函数会将所提供的参数添加上指定的偏移值。比如：

```
int main()
{   std::function<int(int)> offset_42=make_offseter(42);
    std::function<int(int)> offset_123=make_offseter(123);
    std::cout<<offset_42(12)<<","<<offset_123(12)<<std::endl;
    std::cout<<offset_42(12)<<","<<offset_123(12)<<std::endl;
}
```

这段代码会输出54,135两次，因为在第一次调用make_offseter时返回的函数，总是将所提供的参数加 42，而第二次调用 make_offseter 返回的函数，总是将所提供的参数加 123。

这是捕捉局部变量的最安全形式，一切都是被复制的，所以你可以返回一个 lambda 并在原函数作用域之外去调用它。但这并非唯一的选择，相反的，你可以选择通过引用来捕捉所有的东西。在这种情况下，一旦 lambda 所引用的变量因为离开其所在的函数或者语句块作用域而被销毁，调用该 lambda 就成为一种未定义的行为，正如在其他情况下引用一个已经被销毁的变量一样。

以引用的方式捕捉所有局部变量的 lambda 函数使用[&]来引导，如下面的例子所示，

```
int main()
{
    int offset=42;                                                        ←❶
    std::function<int(int)> offset_a=[&](int j){return offset+j;};        ←❷
    offset=123;                                                           ←❸
    std::function<int(int)> offset_b=[&](int j){return offset+j;};        ←❹
    std::cout<<offset_a(12)<<","<<offset_b(12)<<std::endl;                ←❺
    offset=99;                                                            ←❻
    std::cout<<offset_a(12)<<","<<offset_b(12)<<std::endl;                ←❼
}
```

上个例子中，我们在 make_offseter 函数里使用了 lambda 引导符[=]来捕捉偏移量的副本；这个例子的 offset_a 函数使用 lambda 引导符[&]，通过引用来捕捉 offset❷。offset 的初始值是否为 42❶ 并不重要，调用 offset_a(12)的结果总是

取决于offset的当前值。尽管我们是在制造第二个(相同的)lambda函数offset_b❹之前就将offset值改为了123❸，由于第二个lambda还是通过引用进行捕捉的，所以结果取决于offset的当前值。

现在，当我们打印第一行输出的时候❺，offset仍然是123，所以输出是135，135。然而，在第二行输出的地方❼，offset已经被改为99❻，所以这时候的输出是111，111。offset_a和offset_b都是将offset的当前值（99）加到所提供的参数（12）上。

其实，C++之所以称为C++，你并不会受困于这些非黑即白的选项，你可以选择通过复制捕捉一部分参数，然后通过引用捕捉一部分，并且你可以选择仅仅捕捉你显式选定的那些变量，这些都可以通过调整lambda引导符实现。如果你希望复制所有用到的变量，但是有一两个例外，你可以使用lambda引导符的[=]形式，然后在等号后面跟上引用捕捉的变量列表，变量之前冠以&符号。下面的例子将会打印1239，因为i被复制进lambda，而j和k是引用捕捉的。

```
int main()
{
    int i=1234,j=5678,k=9;
    std::function<int()> f=[=,&j,&k]{return i+j+k;};
    i=1;
    j=2;
    k=3;
    std::cout<<f()<<std::endl;
}
```

另外，你可以默认引用捕捉，但是通过复制捕捉变量的一个指定子集。在这种情况下，使用lambda引导符的[&]形式，但在&号后面跟以复制捕捉的变量列表。下面的例子会打印5688，因为i是引用捕捉的，而j和k是复制的。

```
int main()
{
    int i=1234,j=5678,k=9;
    std::function<int()> f=[&,j,k]{return i+j+k;};
    i=1;
    j=2;
    k=3;
    std::cout<<f()<<std::endl;
}
```

如果你只想捕捉提名的变量，你可以省略前面的=或&，仅仅列出需要捕捉的变量，通过前缀&符号来表明引用捕捉而非复制。下面的代码将会打印5682，因为i和k通过引用捕捉，而j是复制的。

```
int main()
{
    int i=1234,j=5678,k=9;
    std::function<int()> f=[&i,j,&k]{return i+j+k;};
```

```
    i=1;
    j=2;
    k=3;
    std::cout<<f()<<std::endl;
}
```

最后这个变化形式允许你确保只有意料之中的变量被捕捉，因为引用任何不在捕捉列表中的局部变量都会导致编译时错误。如果你选择了这个选项，而包含 lambda 的函数又是一个成员函数，你就得小心地访问类成员。

类成员不能被直接捕捉，如果你希望从 lambda 中访问类成员，你必须将 this 指针添加到捕捉列表中先行捕捉。在下面的例子中，lambda 捕捉了 this，以允许访问类成员 some_data。

```
struct X
{
    int some_data;
    void foo(std::vector<int>& vec)
    {
        std::for_each(vec.begin(),vec.end(),
            [this](int& i){i+=some_data;});
    }
};
```

在并发的上下文中，lambda 表达式最大的用处是作为 std::condition_variable::wait()（参见 4.1.1 节）以及 std::packaged_task<>（参见 4.2.1 节）的断言，或是打包小任务的线程池。它们也能作为线程函数（参见 2.1.1 节）传给 std::thread 的构造函数，或是作为使用并行算法的函数，比如 parallel_for_each()（参见 8.5.1 节）。

## A.6　变参模板

变参模板，指的是参数数量可变的模板。正如你一直以来使用的变参函数一样，比如 printf 就可以接受可变数量的参数，你现在有了模板参数数量可变的变参模板。变参模板的使用贯穿整个 C++线程库。比如，std::thread 用来启动线程的构造函数（参见 2.1.1 节）就是一个变参函数模板，std::packaged_task<>（参见 4.2.2 节）是一个变参类模板。从使用者的角度来看，知道这个模板可以接受不限数量的参数就足够了，但如果你想编写一个这样的模板，或者你对它是如何工作的感兴趣，你就需要知道其中的细节。

变参函数是通过函数参数列表中的省略号（...）来声明的，与之类似，变参模板的声明方式，是在模板参数列表中的省略号。

```
template<typename ... ParameterPack>
class my_template
{};
```

你也可以在模板偏特化中使用变参模板，即便是主模板并非变参的。比如，std::packaged_task<>(参见4.2.1节)的主模板只是一个带有单个模板参数的简单模板。

```
template<typename FunctionType>
class packaged_task;
```

然而，该主模板没有在任何地方进行定义，它仅仅是作为偏特化的占位标志。

```
template<typename ReturnType,typename ... Args>
class packaged_task<ReturnType(Args...)>;
```

正是这一偏特化里包含了该类的真正定义。在第4章中，你见过这样的写法，用std::packaged_task<int(std::string, double)>来声明一个在调用时接受std::string和double作为参数的任务，然后通过std::future<int>提供结果。

该声明展示了变参模板的另外两项特性。第一个特性相对简单，你可以在一个声明中同时包含普通的模板参数（比如ReturnType）和变参（Args）。第二个特性演示了在特化的模板参数列表中Args...的用法，显示了当模板实例化时组成Args的类型会在这里列出。事实上由于这是偏特化，它按照模式匹配来工作，在实际的实例化中，此上下文中出现的类型会被捕捉为Args。变参Args称为**参数包**（**parameter pack**），而对Args...的使用称为**包展开**（**pack expansion**）。

与变参函数类似，变参部分既可以是空列表也可以有很多条目。比如，在std::packaged_task<my_class()>中ReturnType参数是my_class，Args参数包是空的，然而在std::packaged_task<void(int, double, my_class&, std::string*)>中ReturnType是void，Args是int、double、my_class&、std::string*的列表。

## 展开参数包

变参模板的强大之处，体现在你如何去进行包的展开，即你并不局限于仅仅将类型列表原样展开。首先，你可以在任何需要类型列表的地方直接使用包展开，比如另一个模板的参数列表。

```
template<typename ... Params>
struct dummy
{
    std::tuple<Params...> data;
};
```

在这个例子中，唯一的成员变量data是std::tuple<>的一个实例，它包含了所有指定的类型，所以dummy<int, double, char>拥有类型为std::tuple<int, double, char>的成员。你可以用普通类型来整合包展开。

```
template<typename ... Params>
struct dummy2
{
    std::tuple<std::string,Params...> data;
};
```

这一次，此元组有一个额外的（第一个）std::string 类型的成员。有趣的部分是你可以使用包展开创建一个模式，它接下来会针对每一个展开的元素复制。你通过将标识包展开的……放到模式的末尾来做到这一点。例如，不是仅创建你的参数包里提供的元素元组，而是可以创建指向这些元素的指针元组，或者甚至指向你的元素的 std::unique_ptr<>元组：

```
template<typename ... Params>
struct dummy3
{
    std::tuple<Params* ...> pointers;
    std::tuple<std::unique_ptr<Params> ...> unique_pointers;
};
```

类型表达式可以如你所想的那样复杂，前提是参数包出现在类型表达式中，并且该表达式后面跟着标识展开的……。当参数包被展开时，包里的每一项会代入类型表达式，以生成结果列表中的相应项。因此，如果你的参数包 Params 包含 int,int,char 的类型，那么 std::tuple<std::pair<std::unique_ptr<Params>,double>...>的展开就是 std::tuple<std::pair<std::unique_ptr<int>,double>,std::pair<std::uniqueptr<int>,double>,std::pair<std::unique_ptr<char>,double>>。如果包展开被用作模板参数列表，该模板无需具有变参，但如果它确实没有，那么包的大小必须恰好匹配模板参数要求的数量。

```
template<typename ... Types>
struct dummy4
{
    std::pair<Types...> data;
};
dummy4<int,char> a;      // ❶ 正确，data 是 std::pair<int,char>
dummy4<int> b;           // ❷ 错误，没有第二个类型
dummy4<int,int,int> c;   // ❸ 错误，类型过多
```

你可以对包展开做的第二件事情，是用它来声明一个函数参数列表。

```
template<typename ... Args>
void foo(Args ... args);
```

这样创建一个新的参数包 args，它是函数参数的列表而不是类型的列表，你可以像之前那样用...来展开它。现在，你可以用带有包展开的模式来声明函数参数，正如你在其他地方使用的展开包模式一样。比如，std::thread 利用它来通过右值引用接受所有的函数参数（参见 A.1 节）：

```
template<typename CallableType,typename ... Args>
thread::thread(CallableType&& func,Args&& ... args);
```

这个函数参数包可以用来通过在被调用函数的参数列表里指定包展开，来调用另一个函数。正如类型展开一样，你可以为结果参数列表里的每个表达式使用一个模式。例如，一个常见的右值引用习惯用法是使用 std::forward<>来保持所提供函数参数的右值性：

```
template<typename ... ArgTypes>
void bar(ArgTypes&& ... args)
{
    foo(std::forward<ArgTypes>(args)...);
}
```

注意在这个例子里，包展开同时包含了类型包 ArgTypes 和函数参数包 args，以及整个表达式后面的省略号。如果你像这样调用 bar：

```
int i;
bar(i,3.141,std::string("hello "));
```

那么展开成为：

```
template<>
void bar<int&,double,std::string>(
    int& args_1,
    double&& args_2,
    std::string&& args_3)
{
    foo(std::forward<int&>(args_1),
        std::forward<double>(args_2),
        std::forward<std::string>(args_3));
}
```

正确地将第一个参数作为左值引用传递给 foo，同时将以右值引用传递另一个参数。

你可以对参数展开做的最后一件事是使用 sizeof...操作符来获取其大小。这是非常简单的：sizeof...(p)就是参数包 p 中的元素个数。无论是类型参数包还是函数参数包，结果都是一样的。这可能是唯一一个你可以使用参数包并且不在后面跟上省略号的情况，省略号已经是 sizeof...运算符的一部分。下面的函数返回提供给它的参数个数：

```
template<typename ... Args>
unsigned count_args(Args ... args)
{
    return sizeof... (Args);
}
```

如同普通的 sizeof 运算符，sizeof...的结果是一个常量表达式，所以它可以被用来指定数组的边界等等。

## A.7 自动推断变量的类型

C++是一门静态类型语言，每个变量的类型在编译时都是已知的。不仅如此，作为

程序员，你还得指定每个变量的类型。在某些情况下，这将导致非常笨拙的名字，例如，

```
std::map<std::string,std::unique_ptr<some_data>> m;
std::map<std::string,std::unique_ptr<some_data>>::iterator
    iter=m.find("my key");
```

传统上，有个解决方案那就是使用 typedef 来减少类型标识符的长度，并有可能消除类型不一致的问题。这在 C++11 中仍然有效，但现在有了一种新的方式，如果一个变量在其声明中初始化自一个相同类型的值，那么你可以将类型指定为 auto。在这种情况下，编译器将自动推断变量的类型与初始化器相同。于是，该迭代器示例可以写作：

```
auto iter=m.find("my key");
```

现在，你不限于只是普通的 auto，你可以修饰它来声明 const 变量、指针或引用变量。这里有几个使用 auto 的变量声明及其相应的变量类型：

```
auto i=42;          // int
auto& j=i;          // int&
auto const k=i;     // int const
auto* const p=&i;   // int * const
```

推断变量类型的规则，基于该语言在其他地方推断类型的规则：函数模板的参数。在形如下面格式的声明中：

```
some-type-expression-involving-auto var=some-expression;
```

var 的类型与用相同表达式推断出的函数模板参数相同，区别在于将 auto 替换成了模板参数的名字：

```
template<typename T>
void f(type-expression var);
f(some-expression);
```

这意味着数组类型衰变到指针和引用都被丢弃，除非该类型表达式显式声明变量为引用，例如：

```
int some_array[45];
auto p=some_array;      // int*
int& r=*p;
auto x=r;               // int
auto& y=r;              // int&
```

这可大大简化变量的声明，特别是完整的类型标识符很长，或者干脆不可知（例如，在一个模板中的函数调用的结果的类型）。

## A.8 线程局部变量

线程局部变量可以在程序中让你为每个线程拥有独立的变量实例。在声明变量时，可以用 thread_local 关键字进行标记，表明它是线程局部的。命名空间范围的变量、

类的静态数据成员和局部变量都可被声明为线程局部的，也就是说，具有**线程存储期**（**thread storage duration**）。

```
thread_local int x;                         ← 命名空间范围内的线程局部变量

class X
{
    static thread_local std::string s;      ← 线程局部的静态类数据成员
};
static thread_local std::string X::s;       ← 需要 X::s 的定义
void foo()
{
    thread_local std::vector<int> v;        ← 线程局部的局部变量
}
```

命名空间范围的线程局部变量和线程局部的静态类数据成员，在相同的翻译单元里的首次使用之前被构造，但并没有指定要提前多久。有些实现方式可能在线程启动时构造线程局部变量，有些则可能在他们被各自线程初次使用之前就立即构造了，还有些可能在其他的时点进行构造，或是根据其用途上下文的某种组合。事实上，如果没有一个来自给定的翻译单元的线程局部变量被使用，则压根就不会保证他们会被构造。这就允许动态加载的模块包含线程局部变量——这些变量可以在来自动态加载模块的线程首次引用线程局部变量的时候被构造。

在函数内声明的线程局部变量，是在给定线程的控制流第一次通过其声明时初始化的。如果不由给定的线程调用该函数，则函数内的任何线程局部变量都不被构造。这一行为与局部静态变量是一样的，区别在于它分别适用于每个线程。

线程局部变量与静态变量共享其他属性——他们在所有进一步初始化（如动态初始化）之前是零初始化的。如果线程本地变量的构造引发异常，则调用 `std::terminate()` 来中止应用程序。

所有在给定线程上构造的线程局部变量的析构函数，运行于在线程函数返回时，与构造函数是相反的顺序。由于初始化的顺序是未指定的，因此确保这种变量的析构函数之间没有相互依存关系是很重要的。如果一个线程局部变量的析构函数以异常退出，`std::terminate()` 会被调用，与构造相类似。

如果一个线程调用 `std::exit()` 或从 `main()` 返回（等效于以 `main()` 的返回值调用 `std::exit()`），那么该线程的线程局部变量也会被销毁。如果任何其他线程在应用程序退出时仍在运行，那些线程的线程局部变量的析构函数不会被调用。

虽然线程局部变量的每个线程上有一个不同的地址，你仍然可以获得指向这种变量的普通指针。该指针将引用拥有该地址的线程中的对象，并可以用来允许其他线程访问该对象。在一个对象被销毁后再去访问它是未定义的行为（一直都是），所以如果将指

向一个线程局部变量的指针传给另一个线程的线程，你需要确保它在其所属线程结束后不被解引用。

## A.9 小结

本附录只是蹭了一下 C++标准引入的新语言特性的皮毛，因为我们只看了那些对Thread 库有积极影响的功能。其他新的语言功能包括静态断言、强类型的枚举、委派构造函数、Unicode 支持、模板别名和一个新的统一的初始化序列，以及一系列较小变化。描述所有新功能的详细信息不在本书的范围之内，可能需要另一本专门的书。目前，对完整的标准变更最好的概览，大概是 Bjarne Stroustrup 的 C++11 常见问题解答[1]，而流行的 C++参考书为了涵盖它会及时地作出修订。

希望本附录中所涵盖的新功能简介提供了足够的深度，以展示它们与 Thread 库有着怎样的关系，并使你能够编写和理解使用这些新功能的多线程的代码。虽然本附录本应为这些特性所涵盖简单用途提供足够的深度，但这仍然只是一份简介，并不是使用这些功能的完整的参考或教程。如果你打算将它们广泛地使用，我建议获取一份这样的参考或教程，以便从中获得最大的益处。

[1] http://www.research.att.com/~bs/C++0xFAQ.html

# 附录B 并发类库简要对比

编程语言和类库对并发与多线程的支持并非新生事物，它们只是最近才在C++标准中得到支持。比如，Java 自发布起就拥有多线程支持，兼容 POSIX 标准的平台提供了用于多线程的 C 语言接口，Erlang 提供了信息传递并发的支持。甚至还有些类似 Boost 一样的 C++库，将各种平台的底层多线程编程接口（POSIX C 接口或其他）进行了封装，具备跨平台的可移植性。

对那些已经有多线程应用编写经验，并且希望利用这些经验来使用新的 C++多线程工具编写代码的人而言，本附录提供了 Java、POSIX C、C++ Boost 线程库与 C++11 中可用工具的对比，同时包括本书中相关章节的交叉引用。

| 功能 | Java | POSIX C | Boost threads | C++11 | 引用章节 |
|---|---|---|---|---|---|
| 启动线程 | java.lang.thread 类 | pthread_t 类型和相关 API 函数：pthread_create()、pthread_detach() 和 pthread_join() | boost::thread 类与成员函数 | std::thread 类与成员函数 | 第 2 章 |
| 互斥 | synchronized 块 | pthread_mutex_t 类型和相关 API 函数：pthread_mutex_lock()、pthread_mutex_unlock()等 | boost::mutex 类与成员函数，boost::lock_guard<>和 boost::unique_lock<>模板 | std::mutex 类与成员函数，std::lock_guard<>和 std::unique_lock<>模板 | 第 3 章 |
| 监控/等待预期 | java.lang.Object 类的 wait()和 notify()方法，在 synchronized 块内使用 | pthread_cond_t 类型与相关 API 函数：pthread_cond_wait()、pthread_cond_timed_wait()等 | boost::condition_variable 和 boost::condition_variable_any 类与成员函数 | std::condition_variable 和 std::condition_variable_any 类与成员函数 | 第 4 章 |
| 原子操作与并发感知内存模型 | volatile 变量，位于 java.util.concurrent.atomic 包中 | 不可用 | 不可用 | std::atomic_xxx 类型，std::atomic<>类模板，std::atomic_thread_fence()函数 | 第 5 章 |
| 线程安全容器 | java.util.concurrent 包中的容器 | 不可用 | 不可用 | 不可用 | 第 6 章和第 7 章 |
| future | java.util.concurrent.future 接口及相关类 | 不可用 | boost::unique_future<>和 boost::shared_future<>类模板 | std::future<>，std::shared_future<>和 std::atomic_future<>类模板 | 第 4 章 |
| 线程池 | java.util.concurrent.ThreadPoolExecutor 类 | 不可用 | 不可用 | 不可用 | 第 9 章 |
| 线程中断 | java.lang.Thread 的 interrupt()方法 | pthread_cancel() | boost::thread 类的 interrupt()成员函数 | 不可用 | 第 9 章 |

# 附录 C　消息传递框架与完整的 ATM 示例

在第 4 章中，我展示了一个在线程之间使用消息传递框架来传递消息的例子，其中简单实现了 ATM 中的代码。下面给出该示例的完整代码，其中包含了消息传递框架。

清单 C.1 展示了消息队列。其中以指向基类的指针存放了一列消息，特定的消息类别使用从该基类派生的类模板来处理。压入一个条目，即是构造一个包装类的相应实例并且存储一个指向它的指针。弹出一个条目，即是返回该指针。由于 `message_base` 类没有任何成员函数，弹出线程在能够访问存储的消息之前，需要将此指针转换为一个合适的 `wrapped_message<T>`指针。

**清单 C.1　简单的消息队列**

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <memory>

namespace messaging
{
    struct message_base
    {
        virtual ~message_base()
        {}
    };
```

我们的队列项目的基类（标注：struct message_base）

```
    template<typename Msg>
    struct wrapped_message:          <-- 每个消息类型有特殊定义
        message_base
    {
        Msg contents;

        explicit wrapped_message(Msg const& contents_):
            contents(contents_)
        {}
    };
                                     我们的消息队列
    class queue                  <--
    {
        std::mutex m;
        std::condition_variable c;
        std::queue<std::shared_ptr<message_base> > q;   <-- 实际的队列存储 message_base 的指针
    public:
        template<typename T>
        void push(T const& msg)
        {
            std::lock_guard<std::mutex> lk(m);
            q.push(std::make_shared<wrapped_message<T> >(msg));   <-- 将发出的消息封装并且存储指针
            c.notify_all();
        }

        std::shared_ptr<message_base> wait_and_pop()
        {
            std::unique_lock<std::mutex> lk(m);
            c.wait(lk,[&]{return !q.empty();});   <-- 阻塞直到队列非空
            auto res=q.front();
            q.pop();
            return res;
        }
    };
}
```

发送消息是通过清单 C.2 所示的 sender 类实例来处理的。它仅仅是对消息队列的简单包装，只允许消息被压入。复制 sender 的实例仅仅复制指向队列的指针，而非队列本身。

**清单 C.2　sender 类**

```
namespace messaging
{
    class sender
    {
        queue*q;        <-- sender 就是封装了队列指针
    public:
        sender():       <-- 默认构造的 sender 没有队列
            q(nullptr)
        {}

        explicit sender(queue*q_):   <-- 允许从指向队列的指针进行构造
            q(q_)
        {}
```

```
        template<typename Message>
        void send(Message const& msg)
        {
            if(q)
            {
                q->push(msg);      在队列上发送推送消息
            }
        }
    };
}
```

接收消息要稍微复杂一点。你不仅要等待队列中的消息，还需要检查其类型是否符合所等待的消息类型，并且调用相应的处理函数。这些都始于清单 C.3 中展示的 receiver 类。

**清单 C.3 receiver 类**

```
namespace messaging
{
    class receiver
    {
        queue q;         一个 receiver 拥有此队列
    public:
        operator sender()      允许隐式转换到引用队列的 sender
        {
            return sender(&q);
        }
        dispatcher wait()      等待队列创建调度器
        {
            return dispatcher(&q);
        }
    };
}
```

与 sender 仅仅引用一个消息队列不同，receiver 拥有它。你可以使用隐式转换来获得一个引用该队列的 sender 类。进行消息调度的复杂性始于对 wait() 的调用，这将创建一个 dispatcher 对象，它从 receiver 中引用该队列。dispatcher 类展示在下一个清单中，正如你所见，这项工作是在析构函数中完成的。在清单 C.4 的例子中，此工作是由等待消息和调度消息组成的。

**清单 C.4 dispatcher 类**

```
namespace messaging
{
    class close_queue      用来关闭队列的消息
    {};

    class dispatcher
    {
```

```
    queue* q;
    bool chained;                                          不能复制的调度器
                                                           实例
    dispatcher(dispatcher const&)=delete;
    dispatcher& operator=(dispatcher const&)=delete;

    template<
        typename Dispatcher,
        typename Msg,                                      允许 TemplateDispatcher 实例访
        typename Func>                                     问内部成员
    friend class TemplateDispatcher;
    void wait_and_dispatch()
    {                                                      ❶ 循环，等待和
        for(;;)                                            调度消息
        {
            auto msg=q->wait_and_pop();
            dispatch(msg);
        }                                                  ❷ dispatch()检查 close_queue 消息，
    }                                                      并抛出

    bool dispatch(
        std::shared_ptr<message_base> const& msg)
    {
        if(dynamic_cast<wrapped_message<close_queue>*>(msg.get()))
        {
            throw close_queue();
        }
        return false;
    }
public:                                                    调度器实例可以
    dispatcher(dispatcher&& other):                        被移动
        q(other.q),chained(other.chained)
    {
        other.chained=true;                                来源不得等待
    }                                                      消息

    explicit dispatcher(queue* q_):
        q(q_),chained(false)
    {}

    template<typename Message,typename Func>               ❸ 使用 TemplateDispatcher
    TemplateDispatcher<dispatcher,Message,Func>            处理特定类型的消息
    handle(Func&& f)
    {
        return TemplateDispatcher<dispatcher,Message,Func>(
            q,this,std::forward<Func>(f));
    }

    ~dispatcher() noexcept(false)                          ❹ 析构函数可能抛出
    {                                                      异常
        if(!chained)
        {
            wait_and_dispatch();
        }
    }
  };
}
```

从 wait()返回的 dispatcher 实例将会被立刻销毁，因为它是临时的，并且如前所述，析构函数承担了这项工作。析构函数调用 wait_and_dispatch()，这是一个等待消息并将其传递给 dispatch()的循环❶。dispatch()本身❷非常简单，它检查消息是否为一个 close_queue 消息，如果是，就抛出一个异常；否则，它返回 false 来指示该消息未被处理。close_queue 异常正是析构函数被标记为 noexcept(false)的原因；如果没有这个注解，析构函数的默认异常规定将会是 noexcept(true)❹，表明没有异常能够被抛出，那么 close_queue 异常就会终止程序。

然而你并非经常去主动调用 wait()，大部分时间你会希望去处理一个消息。这就是 handle()成员函数❸的用武之地。它是一个模板，并且消息类型是无法推断的，所以你必须指定待处理的消息类型，并且传入一个函数（或者可调用的对象）来处理它。handle()自身将队列、当前的 dispatcher 对象和处理函数传递给一个新的 TemplateDispatcher 类模板的实例，来处理指定类型的消息，这些展示在清单 C.5 的例子中。为什么要在等待消息之前就在析构函数里测试 chained 的值呢？因为这样不仅可以防止移入的对象等待消息，而且允许你将等待消息的重任交给新的 TemplateDispatcher 实例。

**清单 C.5 TemplateDispatcher 类模板**

```
namespace messaging
{
    template<typename PreviousDispatcher,typename Msg,typename Func>
    class TemplateDispatcher
    {
        queue* q;
        PreviousDispatcher* prev;
        Func f;
        bool chained;

        TemplateDispatcher(TemplateDispatcher const&)=delete;
        TemplateDispatcher& operator=(TemplateDispatcher const&)=delete;

        template<typename Dispatcher,typename OtherMsg,typename OtherFunc>
        friend class TemplateDispatcher;          <-- TemplateDispatcher 实例之间互为友元
        void wait_and_dispatch()
        {
            for(;;)
            {
                auto msg=q->wait_and_pop();
                if(dispatch(msg))                 <-- ❶ 如果我们处理了消息，跳出循环
                    break;
            }
        }

        bool dispatch(std::shared_ptr<message_base> const& msg)
        {
```

```
            if(wrapped_message<Msg>* wrapper=
                dynamic_cast<wrapped_message<Msg>*>(msg.get()))    <-- ❷ 检查消息类型并调用函数
            {
                f(wrapper->contents);
                return true;
            }
            else
            {
                return prev->dispatch(msg);    <-- ❸ 链至前一个调度器
            }
        }
    public:
        TemplateDispatcher(TemplateDispatcher&& other):
            q(other.q),prev(other.prev),f(std::move(other.f)),
            chained(other.chained)
        {
            other.chained=true;
        }
        TemplateDispatcher(queue* q_,PreviousDispatcher* prev_,Func&& f_):
            q(q_),prev(prev_),f(std::forward<Func>(f_)),chained(false)
        {
            prev_->chained=true;
        }
        template<typename OtherMsg,typename OtherFunc>
        TemplateDispatcher<TemplateDispatcher,OtherMsg,OtherFunc>
        handle(OtherFunc&& of)    <-- ❹ 可以链接附加的处理函数
        {
            return TemplateDispatcher<
                TemplateDispatcher,OtherMsg,OtherFunc>(
                    q,this,std::forward<OtherFunc>(of));
        }
        ~TemplateDispatcher() noexcept(false)    <-- ❺ 析构函数又是noexcept(false)的
        {
            if(!chained)
            {
                wait_and_dispatch();
            }
        }
    };
}
```

TemplateDispatcher<>类模板是基于 dispatcher 类构建的，并且二者几乎雷同。尤其是析构函数仍然调用了 wait_and_dispatch()来等待一个消息。

如果你处理了消息，那么就不会抛出异常，所以你需要在消息循环❶中检查你是否真的处理了消息。当你成功处理了消息时，消息处理即行停止，你就可以等待下一时刻的另一组消息。如果你恰好得到了一条匹配的消息类型，那么所提供的函数就会被调用 ❷，而不是抛出异常（尽管处理函数自身可能会抛出异常）。如果没有得到匹配的消息，那么就链接至前一个 dispatcher❸。在首个实例中，它就是 dispatcher，但如果你将调用链接至 handle() 函数 ❹，以允许多种类型的消息被处理，这可能会先

于 TemplateDispatcher<>初始化，如果消息不匹配的话，这将会反过来链接到前一个句柄上。因为所有句柄都可能引发异常（包括 dispatcher 为 close_queue 消息的默认句柄），析构函数必须再次声明为 noexcept(false)❺。

这个简单的框架允许你将任意类型的消息压入队列中，然后在接收端有选择地匹配你能够处理的消息。它同时允许你为了压入消息而绕过对队列的引用，同时保持接收端的私密性。

为了完成第 4 章中的示例，在清单 C.6 中给出了消息，清单 C.7、清单 C.8 和清单 C.9 中给出几种状态机，驱动代码在清单 C.10 中给出。

**清单 C.6 ATM 消息**

```
struct withdraw
{
    std::string account;
    unsigned amount;
    mutable messaging::sender atm_queue;
    withdraw(std::string const& account_,
             unsigned amount_,
             messaging::sender atm_queue_):
        account(account_),amount(amount_),
        atm_queue(atm_queue_)
    {}
};
struct withdraw_ok
{};
struct withdraw_denied
{};
struct cancel_withdrawal
{
    std::string account;
    unsigned amount;
    cancel_withdrawal(std::string const& account_,
                      unsigned amount_):
        account(account_),amount(amount_)
    {}
};
struct withdrawal_processed
{
    std::string account;
    unsigned amount;
    withdrawal_processed(std::string const& account_,
                         unsigned amount_):
        account(account_),amount(amount_)
    {}
};
```

```
struct card_inserted
{
    std::string account;

    explicit card_inserted(std::string const& account_):
        account(account_)
    {}

};
struct digit_pressed
{
    char digit;

    explicit digit_pressed(char digit_):
        digit(digit_)
    {}

};
struct clear_last_pressed
{};
struct eject_card
{};
struct withdraw_pressed
{
    unsigned amount;

    explicit withdraw_pressed(unsigned amount_):
        amount(amount_)
    {}

};
struct cancel_pressed
{};
struct issue_money
{
    unsigned amount;
    issue_money(unsigned amount_):
        amount(amount_)
    {}
};
struct verify_pin
{
    std::string account;
    std::string pin;
    mutable messaging::sender atm_queue;

    verify_pin(std::string const& account_,std::string const& pin_,
            messaging::sender atm_queue_):
        account(account_),pin(pin_),atm_queue(atm_queue_)
    {}
};
struct pin_verified
{};
```

```
struct pin_incorrect
{};

struct display_enter_pin
{};

struct display_enter_card
{};

struct display_insufficient_funds
{};

struct display_withdrawal_cancelled
{};

struct display_pin_incorrect_message
{};

struct display_withdrawal_options
{};

struct get_balance
{
    std::string account;
    mutable messaging::sender atm_queue;

    get_balance(std::string const& account_,messaging::sender atm_queue_):
        account(account_),atm_queue(atm_queue_)
    {}
};

struct balance
{
    unsigned amount;

    explicit balance(unsigned amount_):
        amount(amount_)
    {}
};

struct display_balance
{
    unsigned amount;
    explicit display_balance(unsigned amount_):
        amount(amount_)
    {}
};

struct balance_pressed
{};
```

清单 C.7 ATM 状态机

```
class atm
{
    messaging::receiver incoming;
    messaging::sender bank;
    messaging::sender interface_hardware;
    void (atm::*state)();
```

```
std::string account;
unsigned withdrawal_amount;
std::string pin;

void process_withdrawal()
{
    incoming.wait()
        .handle<withdraw_ok>(
            [&](withdraw_ok const& msg)
            {
                interface_hardware.send(
                    issue_money(withdrawal_amount));
                bank.send(
                    withdrawal_processed(account,withdrawal_amount));
                state=&atm::done_processing;
            }
            )
        .handle<withdraw_denied>(
            [&](withdraw_denied const& msg)
            {
                interface_hardware.send(display_insufficient_funds());
                state=&atm::done_processing;
            }
            )
        .handle<cancel_pressed>(
            [&](cancel_pressed const& msg)
            {
                bank.send(
                    cancel_withdrawal(account,withdrawal_amount));
                interface_hardware.send(
                    display_withdrawal_cancelled());
                state=&atm::done_processing;
            }
            );
}
void process_balance()
{
    incoming.wait()
        .handle<balance>(
            [&](balance const& msg)
            {
                interface_hardware.send(display_balance(msg.amount));
                state=&atm::wait_for_action;
            }
            )
        .handle<cancel_pressed>(
            [&](cancel_pressed const& msg)
            {
                state=&atm::done_processing;
            }
            );
}

void wait_for_action()
{
```

```
    interface_hardware.send(display_withdrawal_options());
    incoming.wait()
        .handle<withdraw_pressed>(
            [&](withdraw_pressed const& msg)
            {
                withdrawal_amount=msg.amount;
                bank.send(withdraw(account,msg.amount,incoming));
                state=&atm::process_withdrawal;
            }
            )
        .handle<balance_pressed>(
            [&](balance_pressed const& msg)
            {
                bank.send(get_balance(account,incoming));
                state=&atm::process_balance;
            }
            )
        .handle<cancel_pressed>(
            [&](cancel_pressed const& msg)
            {
                state=&atm::done_processing;
            }
            );
}
void verifying_pin()
{
    incoming.wait()
        .handle<pin_verified>(
            [&](pin_verified const& msg)
            {
                state=&atm::wait_for_action;
            }
            )
        .handle<pin_incorrect>(
            [&](pin_incorrect const& msg)
            {
                interface_hardware.send(
                    display_pin_incorrect_message());
                state=&atm::done_processing;
            }
            )
        .handle<cancel_pressed>(
            [&](cancel_pressed const& msg)
            {
                state=&atm::done_processing;
            }
            );
}
void getting_pin()
{
    incoming.wait()
        .handle<digit_pressed>(
            [&](digit_pressed const& msg)
```

```
                {
                    unsigned const pin_length=4;
                    pin+=msg.digit;
                    if(pin.length()==pin_length)
                    {
                        bank.send(verify_pin(account,pin,incoming));
                        state=&atm::verifying_pin;
                    }
                }
                )
            .handle<clear_last_pressed>(
                [&](clear_last_pressed const& msg)
                {
                    if(!pin.empty())
                    {
                        pin.pop_back();
                    }
                }
                )
            .handle<cancel_pressed>(
                [&](cancel_pressed const& msg)
                {
                    state=&atm::done_processing;
                }
                );
    }
    void waiting_for_card()
    {
        interface_hardware.send(display_enter_card());
        incoming.wait()
            .handle<card_inserted>(
                [&](card_inserted const& msg)
                {
                    account=msg.account;
                    pin="";
                    interface_hardware.send(display_enter_pin());
                    state=&atm::getting_pin;
                }
                );
    }
    void done_processing()
    {
        interface_hardware.send(eject_card());
        state=&atm::waiting_for_card;
    }
    atm(atm const&)=delete;
    atm& operator=(atm const&)=delete;
public:
    atm(messaging::sender bank_,
        messaging::sender interface_hardware_):
        bank(bank_),interface_hardware(interface_hardware_)
    {}
```

```
    void done()
    {
        get_sender().send(messaging::close_queue());
    }

    void run()
    {
        state=&atm::waiting_for_card;
        try
        {
            for(;;)
            {
                (this->*state)();
            }
        }
        catch(messaging::close_queue const&)
        {
        }
    }

    messaging::sender get_sender()
    {
        return incoming;
    }
};
```

清单 C.8 银行状态机

```
class bank_machine
{
    messaging::receiver incoming;
    unsigned balance;
public:
    bank_machine():
        balance(199)
    {}

    void done()
    {
        get_sender().send(messaging::close_queue());
    }

    void run()
    {
        try
        {
            for(;;)
            {
                incoming.wait()
                    .handle<verify_pin>(
                        [&](verify_pin const& msg)
                        {
                            if(msg.pin=="1937")
                            {
                                msg.atm_queue.send(pin_verified());
                            }
```

```
                    else
                    {
                        msg.atm_queue.send(pin_incorrect());
                    }
                }
                )
            .handle<withdraw>(
                [&](withdraw const& msg)
                {
                    if(balance>=msg.amount)
                    {
                        msg.atm_queue.send(withdraw_ok());
                        balance-=msg.amount;
                    }
                    else
                    {
                        msg.atm_queue.send(withdraw_denied());
                    }
                }
                )
            .handle<get_balance>(
                [&](get_balance const& msg)
                {
                    msg.atm_queue.send(::balance(balance));
                }
                )
            .handle<withdrawal_processed>(
                [&](withdrawal_processed const& msg)
                {
                }
                )
            .handle<cancel_withdrawal>(
                [&](cancel_withdrawal const& msg)
                {
                }
                );
        }
    }
    catch(messaging::close_queue const&)
    {
    }
  }

  messaging::sender get_sender()
  {
    return incoming;
  }
};
```

清单 C.9　用户界面状态机

```
class interface_machine
{
    messaging::receiver incoming;
public:
```

```
void done()
{
    get_sender().send(messaging::close_queue());
}
void run()
{
    try
    {
        for(;;)
        {
            incoming.wait()
                .handle<issue_money>(
                    [&](issue_money const& msg)
                    {
                        {
                            std::lock_guard<std::mutex> lk(iom);
                            std::cout<<"Issuing "
                                     <<msg.amount<<std::endl;
                        }
                    }
                    )
                .handle<display_insufficient_funds>(
                    [&](display_insufficient_funds const& msg)
                    {
                        {
                            std::lock_guard<std::mutex> lk(iom);
                            std::cout<<"Insufficient funds"<<std::endl;
                        }
                    }
                    )
                .handle<display_enter_pin>(
                    [&](display_enter_pin const& msg)
                    {
                        {
                            std::lock_guard<std::mutex> lk(iom);
                            std::cout
                                <<"Please enter your PIN (0-9)"
                                <<std::endl;
                        }
                    }
                    )
                .handle<display_enter_card>(
                    [&](display_enter_card const& msg)
                    {
                        {
                            std::lock_guard<std::mutex> lk(iom);
                            std::cout<<"Please enter your card (I)"
                                     <<std::endl;
                        }
                    }
                    )
                .handle<display_balance>(
                    [&](display_balance const& msg)
                    {
```

```
                {
                    std::lock_guard<std::mutex> lk(iom);
                    std::cout
                        <<"The balance of your account is "
                        <<msg.amount<<std::endl;
                }
            }
            )
        .handle<display_withdrawal_options>(
            [&](display_withdrawal_options const& msg)
            {
                {
                    std::lock_guard<std::mutex> lk(iom);
                    std::cout<<"Withdraw 50? (w)"<<std::endl;
                    std::cout<<"Display Balance? (b)"
                        <<std::endl;
                    std::cout<<"Cancel? (c)"<<std::endl;
                }
            }
            )
        .handle<display_withdrawal_cancelled>(
            [&](display_withdrawal_cancelled const& msg)
            {
                {
                    std::lock_guard<std::mutex> lk(iom);
                    std::cout<<"Withdrawal cancelled"
                        <<std::endl;
                }
            }
            )
        .handle<display_pin_incorrect_message>(
            [&](display_pin_incorrect_message const& msg)
            {
                {
                    std::lock_guard<std::mutex> lk(iom);
                    std::cout<<"PIN incorrect"<<std::endl;
                }
            }
            )
        .handle<eject_card>(
            [&](eject_card const& msg)
            {
                {
                    std::lock_guard<std::mutex> lk(iom);
                    std::cout<<"Ejecting card"<<std::endl;
                }
            }
            );
        }
    }
    catch(messaging::close_queue&)
    {
    }
}
```

```
    messaging::sender get_sender()
    {
        return incoming;
    }
};
```

清单 C.10 驱动代码

```
int main()
{
    bank_machine bank;
    interface_machine interface_hardware;

    atm machine(bank.get_sender(),interface_hardware.get_sender());

    std::thread bank_thread(&bank_machine::run,&bank);
    std::thread if_thread(&interface_machine::run,&interface_hardware);
    std::thread atm_thread(&atm::run,&machine);
    messaging::sender atmqueue(machine.get_sender());

    bool quit_pressed=false;

    while(!quit_pressed)
    {
        char c=getchar();
        switch(c)
        {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            atmqueue.send(digit_pressed(c));
            break;
        case 'b':
            atmqueue.send(balance_pressed());
            break;
        case 'w':
            atmqueue.send(withdraw_pressed(50));
            break;
        case 'c':
            atmqueue.send(cancel_pressed());
            break;
        case 'q':
            quit_pressed=true;
            break;
        case 'i':
            atmqueue.send(card_inserted("acc1234"));
            break;
        }
    }
```

```
    bank.done();
    machine.done();
    interface_hardware.done();
    atm_thread.join();
    bank_thread.join();
    if_thread.join();
}
```

# 附录 D　C++线程类库参考

## D.1 <chrono>头文件

<chrono>头文件提供了表示时间点和 duration 的类，以及可作为 time_point 源的时钟类。每个时钟都有一个 is_steady 静态数据成员，能够指示该时钟是否为一个按照一致的速率（并且不能被调整）行进的匀速时钟。std::chrono::steady_clock 类是唯一一个确保匀速的时钟。

**头文件内容**

```
namespace std
{
   namespace chrono
   {
      template<typename Rep,typename Period = ratio<1>>
      class duration;
      template<
         typename Clock,
         typename Duration = typename Clock::duration>
      class time_point;
      class system_clock;
      class steady_clock;
      typedef unspecified-clock-type high_resolution_clock;
   }
}
```

## D.1.1 std::chrono::duration 类模板

std::chrono::duration 类模板提供了一个代表时间段的工具。模板的参数 Rep 和 Period 是用来存储时间段值的数据类型，还有一个 std::ratio 类模板的实例用来指示与下一个时钟周期之间的时间长度（单位是几分之一秒）。因此 std::chrono::duration<int, std::milli>是以 int 类型值存储的毫秒数，std::chrono::duration<short, std::ratio<1, 50>>是以 short 值类型存储的 50 分之一秒的数量，std::chrono::duration<long long, std::ration<60, 1>>是以 long long 值类型存储的分钟数。

**类定义**

```
template <class Rep, class Period=ratio<1> >
class duration
{
public:
    typedef Rep rep;
    typedef Period period;

    constexpr duration() = default;
    ~duration() = default;

    duration(const duration&) = default;
    duration& operator=(const duration&) = default;

    template <class Rep2>
    constexpr explicit duration(const Rep2& r);

    template <class Rep2, class Period2>
    constexpr duration(const duration<Rep2, Period2>& d);

    constexpr rep count() const;
    constexpr duration operator+() const;
    constexpr duration operator-() const;
    duration& operator++();
    duration operator++(int);
    duration& operator--();
    duration operator--(int);
    duration& operator+=(const duration& d);
    duration& operator-=(const duration& d);
    duration& operator*=(const rep& rhs);
    duration& operator/=(const rep& rhs);
    duration& operator%=(const rep& rhs);
    duration& operator%=(const duration& rhs);
    static constexpr duration zero();
    static constexpr duration min();
    static constexpr duration max();
};

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
    const duration<Rep1, Period1>& lhs,
```

```
    const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);

template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);

template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

**需求**

Rep 必须是内置的数值类型，或者是类似数字的用户定义类型。Period 必须是 std::ratio<>的一个实例。

### std::chrono::duration::rep typedef

该 typedef 定义的类型用于保存 duration 值中的刻度数目。

**声明**

```
typedef Rep rep;
```

rep 是一个用来存放 duration 对象的内部表示值的类型。

### std::chrono::duration::period typedef

该 typedef 定义了一个 std::ratio 类模板实例，该实例规定了时间段的计数值的单位是多少分之一秒。例如，如果 peroid 是 std::ratio<1, 50>，那么一个 count() 为 $N$ 的时间段值表示 50 分之 $N$ 秒。

**声明**

```
typedef Period period;
```

### std::chrono::duration 默认构造函数

用默认值构造 std::chrono::duration 实例。

**声明**

```
constexpr duration() = default;
```

**结果**

duration（类型为 rep）的内部值被默认初始化。

### std::chrono::duration 来自计数值的转换构造函数

用指定的计数值构造一个 std::chrono::duration 实例。

**声明**

```
template <class Rep2>
constexpr explicit duration(const Rep2& r);
```

**结果**

duration 对象的内部值被初始化为 static_cast<rep>(r)。

**需求**

只有当 Rep2 可以隐式转换为 Rep，并且 Rep 是浮点类型或者 Rep2 不是浮点类型的时候，此构造函数才会参与到重载方案中。

**后置条件**

```
this->count()==static_cast<rep>(r)
```

### std::chrono::duration 来自另一个 STD::CHRONO::DURATION 值的转换构造函数

通过比例缩放另一个 std::chrono::duration 对象的计数值，构造 std::chrono::duration 实例。

**声明**

```
template <class Rep2, class Period2>
constexpr duration(const duration<Rep2,Period2>& d);
```

**结果**

用 duration_cast<duration<Rep, Period>>(d).count() 来初始化 duration 对象的内部值。

**需求**

只有当 Rep 是浮点类型，或者 Rep2 不是一个浮点类型并且 Period2 是 Peroid 的整数倍时（即 ratio_divide<Period2, Period>::den ==1），此构造函数才会参与到重载方案中。这样可以在将一个较短的时间存储到代表较长时间间隔的变量

时，避免意料之外的截断（同时导致精度的损失）。

后置条件

```
this->count()==duration_cast<duration<Rep,Period>>(d).count()
```

示例

```
duration<int,ratio<1,1000>> ms(5);        <-- 5 毫秒
duration<int,ratio<1,1>> s(ms);           <-- 错误：不能将 ms 存储为整数秒
duration<double,ratio<1,1>> s2(ms);       <-- 正确，s2.count==0.005
duration<int,ratio<1,1000000>> us(ms);    <-- 正确，us.count==5000
```

## std::chrono::count 成员函数

获取时间段值。

声明

```
constexpr rep count() const;
```

返回

duration 对象的内部值，以 rep 类型表示。

## std::chrono::duration::operator+一元加号运算符

这其实没有运算：仅返回*this 的副本。

声明

```
constexpr duration operator+() const;
```

返回

```
*this
```

## std::chrono::duration::operator-一元减号运算符

返回一个时间段值，其 count()值是 this->count()的负值。

声明

```
constexpr duration operator-() const;
```

返回

```
duration(-this->count());
```

## std::chrono::duration:operator++前置自增运算符

增加内部计数。

声明

```
duration& operator++();
```

结果

```
++this->internal_count;
```

返回

```
*this
```

### std::chrono::duration::operator++后置自增运算符

增加内部计数，并且返回增加前的`*this`值。

声明

```
duration operator++(int);
```

结果

```
duration temp(*this);
++(*this);
return temp;
```

### std::chrono::duration::operator--前置自减运算符

减小内部计数。

声明

```
duration& operator--();
```

结果

```
--this->internal_count;
```

返回

```
*this
```

### std::chrono::duration::operator--后置自减运算符

减小内部计数，并且返回减小前的`*this`值。

声明

```
duration operator--(int);
```

结果

```
duration temp(*this);
--(*this);
return temp;
```

### std::chrono::duration::operator+=复合赋值运算符

将另一个 duration 对象的计数值加到*this 的内部计数值上。

**声明**

```
duration& operator+=(duration const& other);
```

**结果**

```
internal_count+=other.count();
```

**返回**

```
*this
```

### std::chrono::duration::operator-=复合赋值运算符

从*this 内部的计数值减去另一个 duration 对象的计数值。

**声明**

```
duration& operator-=(duration const& other);
```

**结果**

```
internal_count-=other.count();
```

**返回**

```
*this
```

### std::chrono::duration::operator*=复合赋值运算符

将*this 内部计数值乘以给定的数值。

**声明**

```
duration& operator*=(rep const& rhs);
```

**结果**

```
internal_count*=rhs;
```

**返回**

```
*this
```

### std::chrono::duration::operator/=复合赋值运算符

将*this 的内部计数值除以给定的数值。

**声明**

```
duration& operator/=(rep const& rhs);
```

**结果**

```
internal_count/=rhs;
```

**返回**

```
*this
```

### std::chrono::duration::operator%=复合赋值运算符

将*this 内部计数值设为其与给定数值相除的余数。

**声明**

```
duration& operator%=(rep const& rhs);
```

**结果**

```
internal_count%=rhs;
```

**返回值**

```
*this
```

### std::chrono::duration::operator%=复合赋值运算符

将*this 内部计数值设为其与另一 duration 对象计数值相除的余数。

**声明**

```
duration& operator%=(duration const& rhs);
```

**结果**

```
internal_count%=rhs.count();
```

**返回值**

```
*this
```

### std::chrono::duration::zero 静态成员函数

返回一个表示零值的 duration 对象。

**声明**

```
constexpr duration zero();
```

返回值

```
duration(duration_values<rep>::zero());
```

### std::chrono::duration::min 静态成员函数

返回一个 duration 对象，该对象保存着给定实例最小的可能值。

声明

```
constexpr duration min();
```

返回值

```
duration(duration_values<rep>::min());
```

### std::chrono::duration::max 静态成员函数

返回一个 duration 对象，该对象保存着给定实例最大的可能值。

声明

```
constexpr duration max();
```

返回值

```
duration(duration_values<rep>::max());
```

### std::chrono::duration 相等比较运算符

比较两个 duration 对象是否相等，即便它们具有不同的表现形式和周期。

声明

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

需求

每个 lhs 必须可以隐式地转换到 rhs，反之亦然。如果它们不能彼此进行隐式转换，或者它们是不同的 duration 实例化结果但能够相互隐式转换，该表达式都是不规范的。

结果

如果 CommonDuration 是 std::common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type 的同义词，那么 lhs==rhs 返回 CommonDuration(lhs).count()==CommonDuration(rhs).count() 的结果。

### std::chrono::duration 不等比较运算符

比较两个 duration 对象是否不相等，即便它们具有不同的表现形式和/或周期。

**声明**

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

**需求**

每个 lhs 必须可以隐式地转换到 rhs，反之亦然。如果它们不能彼此进行隐式转换，或者它们是不同的 duration 实例化结果但能够相互隐式转换，该表达式都是不规范的。

**返回**

```
!(lhs==rhs)
```

### std::chrono::duration 小于比较运算符

比较两个 duration 对象，看其中一个是否小于另一个，即便它们具有不同的表现形式和/或周期。

**声明**

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

**需求**

每个 lhs 必须可以隐式地转换到 rhs，反之亦然。如果它们不能彼此进行隐式转换，或者它们是不同的 duration 实例化结果但能够相互隐式转换，该表达式都是不规范的。

**结果**

如果 CommonDuration 是 std::common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type 的同义词，那么 lhs<rhs 返回 CommonDuration(lhs).count()<CommonDuration(rhs).count() 的结果。

### std::chrono::duration 大于比较运算符

比较两个 duration 对象，看其中一个是否大于另一个，即便它们具有不同的表现形式和/或周期。

**声明**

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

**需求**

每个 lhs 必须可以隐式地转换到 rhs，反之亦然。如果它们不能彼此进行隐式转换，或者它们是不同的 duration 实例化结果但能够相互隐式转换，该表达式都是不规范的。

**返回**

```
rhs<lhs
```

## std::chrono::duration 小于或等于比较运算符

比较两个 duration 对象，看其中一个是否小于或者等于另一个，即便它们具有不同的表现形式和/或周期。

**声明**

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

**需求**

每个 lhs 必须可以隐式地转换到 rhs，反之亦然。如果它们不能彼此进行隐式转换，或者它们是不同的 duration 实例化结果但能够相互隐式转换，该表达式都是不规范的。

**返回**

```
!(rhs<lhs)
```

## std::chrono::duration 大于或等于比较运算符

比较两个 duration 对象，看其中一个是否大于或者等于另一个，即便它们具有不同的表现形式和周期。

**声明**

```
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(
    const duration<Rep1, Period1>& lhs,
    const duration<Rep2, Period2>& rhs);
```

**需求**

每个 lhs 必须可以隐式地转换到 rhs，反之亦然。如果它们不能彼此进行隐式转换

换，或者它们是不同的 duration 实例化结果但能够相互隐式转换，该表达式都是不规范的。

**返回**

```
!(lhs<rhs)
```

### std::chrono::duration_cast 非成员函数

显式地将一个 std::chrono::duration 对象转换为指定的 std::chrono::duration 实例。

**声明**

```
template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

**需求**

ToDuration 必须是 std::chrono::duration 的实例。

**返回**

时间段 d 被转换为 ToDuration 所指定的时间段类型。这种方法能够尽量减少不同尺度和表示类型之间的转换所造成的精度损失。

## D.1.2 std::chrono::time_point 类模板

std::chrono::time_point 类模板表示一个以特定时钟来计量的时间点。它被指定为特定时钟所经过的一段纪元（**epoch**）。模板参数 Clock 指定了这一时钟（每个不同的时钟必须具备单独的类型），同时 Duration 模板参数是用来计量经过时间的类型，且必须是 std::chrono::duration 类模板的实例。Duration 默认情况下是 Clock 的默认时间段类型。

**类定义**

```
template <class Clock,class Duration = typename Clock::duration>
class time_point
{
public:
    typedef Clock clock;
    typedef Duration duration;
    typedef typename duration::rep rep;
    typedef typename duration::period period;

    time_point();
    explicit time_point(const duration& d);

    template <class Duration2>
    time_point(const time_point<clock, Duration2>& t);

    duration time_since_epoch() const;
```

```
    time_point& operator+=(const duration& d);
    time_point& operator-=(const duration& d);

    static constexpr time_point min();
    static constexpr time_point max();
};
```

### std::chrono::time_point 默认构造函数

构造一个 time_point，表示相关联的 Clock 的间隔；其内部的时间段被初始化为 Duration::zero()。

**声明**

```
time_point();
```

**后置条件**

对于新的默认构造 time_point 对象 tp，tp.time_since_epoch() == tp::duration::zero()。

### std::chrono::time_point 时间段构造函数

构造 time_point，表示其相关联的 Clock 的间隔为指定的时间段。

**声明**

```
explicit time_point(const duration& d);
```

**后置条件**

对于 time_point 对象 tp，以表示时间段 d 的 tp(d) 进行构造，tp.time_since_epoch() == d。

### std::chrono::time_point 转换构造函数

从另一个具有相同 Clock 但不同的 Duration 的 time_point 对象，来构造一个 time_point 对象。

**声明**

```
template <class Duration2>
time_point(const time_point<clock, Duration2>& t);
```

**需求**

Duration2 必须能够隐式地转换为 Duration。

**结果**

例如 time_point(t.time_since_epoch())。t.time_since_epoch()的返回值被隐式转换为 Duration 类型的对象，且该值被存储在新构造的 time_point 对象中。

### std::chrono::time_point::time_since_epoch 成员函数

获取特定 time_point 对象的时钟间隔时间段。

**声明**

```
duration time_since_epoch() const;
```

**返回**

*this 中存储的 duration 值。

### std::chrono::time_point::operator+=复合赋值运算符

将指定的 duration 加到指定的 time_point 对象所存储的值上。

**声明**

```
time_point& operator+=(const duration& d);
```

**结果**

将 d 加到*this 内部的 duration 对象上，例如：

```
this->internal_duration += d;
```

**返回**

```
*this
```

### std::chrono::time_point::operator-=复合赋值运算符

将指定的 time_point 对象中存储的值中减去指定的 duration。

**声明**

```
time_point& operator-=(const duration& d);
```

**结果**

从*this 内部的 duration 对象减去 d，例如：

```
this->internal_duration -= d;
```

**返回**

```
*this
```

### std::chrono::time_point::min 静态成员函数

得到一个代表其类型的最小可能值的 time_point 对象。

**声明**

```
static constexpr time_point min();
```

返回

time_point(time_point::duration::min())

### td::chrono::time_point::max 静态成员函数

得到一个代表其类型的最大可能值的 time_point 对象。

声明

```
static constexpr time_point max();
```

返回

time_point(time_point::duration::max())

## D.1.3 std::chrono::system_clock 类

std::chrono::system_clock 类提供了从系统真实时间获取当前挂钟时间的方法。当前时间可以通过调用 std::chrono::system_clock::now() 来获得。std::chrono::system_clock::time_point 的实例可以通过 std::chrono::system_clock::to_time_t() 和 std::chrono::system_clock::to_ time_point() 函数，而与 time_t 相互转换。系统时钟不是匀速的，所以之后对 std::chrono::system_clock::now() 的调用，可能会返回比之前的调用更早的时间（比如，操作系统的时钟被手动调整过或是与外部时钟进行了同步）。

类定义

```
class system_clock
{
public:
    typedef unspecified-integral-type rep;
    typedef std::ratio<unspecified,unspecified> period;
    typedef std::chrono::duration<rep,period> duration;
    typedef std::chrono::time_point<system_clock> time_point;
    static const bool is_steady=unspecified;

    static time_point now() noexcept;

    static time_t to_time_t(const time_point& t) noexcept;
    static time_point from_time_t(time_t t) noexcept;
};
```

### std::chrono::system_clock::rep typedef

某个整数类型的类型别名，用来存储 duration 值中的刻度数量。

声明

```
typedef unspecified-integral-type rep;
```

### std::chrono::system_clock::periodtypedef

某 std::ratio 类模板实例的类型别名，用来指定在两个 duration 或 time_point 间最小的秒数（或几分之一秒）。period 指定了时钟的精度，而非其步进频率。

**声明**

```
typedef std::ratio<unspecified,unspecified> period;
```

### std::chrono::system_clock::durationtypedef

std::chrono::duration 类模板的实例，它能够保存由系统范围实时时钟所返回的任意两个时间点的差值。

**声明**

```
typedef std::chrono::duration<
    std::chrono::system_clock::rep,
    std::chrono::system_clock::period> duration;
```

### std::chrono::system_clock::time_pointtypedef

std::chrono::time_point 类模板的实例，它能够保存由系统范围实时时钟返回的时间点。

**声明**

```
typedef std::chrono::time_point<std::chrono::system_clock> time_point;
```

### std::chrono::system_clock::now 静态成员函数

从系统范围实时时钟获取当前的挂钟时间。

**声明**

```
time_point now() noexcept;
```

**返回**

代表当前系统范围实时时钟的当前时间的 time_point。

**引发**

如果发生错误，引发 std::system_error 类型的异常。

### std::chrono::system_clock::to_time_t 静态成员函数

将 time_point 实例转换到 time_t。

声明

```
time_t to_time_t(time_point const& t) noexcept;
```

返回

表示与t相同时间点的time_t值，t会被进位或截取至秒的精度。

引发

如果发生错误，引发std::system_error类型的异常。

### std::chrono::system_clock::from_time_t 静态成员函数

将time_t实例转换到time_point。

声明

```
time_point from_time_t(time_t const& t) noexcept;
```

返回

代表与t相同时间点的time_point值。

引发

如果发生错误，引发std::system_error类型的异常。

## D.1.4 std::chrono::steady_clock 类

std::chrono::steady_clock类提供了对系统范围匀速时钟的访问。当前时间可以通过调用 std::chrono::steady_clock::now() 来获取。在std::chrono::steady_clock::now()和挂钟时间之间并没有固定的关系。匀速的时钟不能往回走，所以如果一个对 std::chrono::steady clock::now()的调用在另一个对它的调用之前，那么第二个调用必须返回与第一个相同或比它更晚的时间点。此时钟以统一的速率，尽可能久地前进。

类定义

```
class steady_clock
{
public:
    typedef unspecified-integral-type rep;
    typedef std::ratio<
        unspecified,unspecified> period;
    typedef std::chrono::duration<rep,period> duration;
    typedef std::chrono::time_point<steady_clock>
        time_point;
    static const bool is_steady=true;

    static time_point now() noexcept;
};
```

### std::chrono::steady_clock::rep typedef

某个整数类型的类型别名，用来存储 duration 值中的刻度数量。

**声明**

```
typedef unspecified-integral-type rep;
```

### std::chrono::steady_clock::period typedef

某 std::ratio 类模板实例的类型别名，用来指定在两个 duration 或 time_point 间最小的秒数（或几分之一秒）。period 指定了时钟的精度，而非其步进频率。

**声明**

```
typedef std::ratio<unspecified,unspecified> period;
```

### std::chrono::steady_clock::duration typedef

std::chrono::duration 类模板的实例，它能够保存由系统范围匀速的时钟所返回的任意两个时间点的差值。

**声明**

```
typedef std::chrono::duration<
    std::chrono::steady_clock::rep,
    std::chrono::steady_clock::period> duration;
```

### std::chrono::steady_clock::time_pointtypedef

std::chrono::time_point 类模板的实例，它能够保存由系统范围匀速的时钟返回的时间点。

**声明**

```
typedef std::chrono::time_point<std::chrono::steady_clock> time_point;
```

### std::chrono::steady_clock::now 静态成员函数

从系统范围匀速的时钟获取当前的时间。

**声明**

```
time_point now() noexcept;
```

**返回**

代表当前系统范围匀速时钟的当前时间的 time_point。

**引发**

如果发生错误，引发 std::system_error 类型的异常。

同步

如果一个对 std:;chrono::steady_clock::now()的调用发生在另一个之前，那么第一个调用所返回的 time_point 必须小于或等于第二个调用所返回的值。

## D.1.5 std::chrono::high_resolution_clock typedef

std::chrono::high_resolution_clock 类以高精度提供了对系统范围时钟的访问。对于所有的时钟，当前时间可以通过调用 std::chrono::high_resolution_clock::now()来获取。std::chrono::high_resolution_clock 可以是 std::chrono::system_clock 类或者是 std::chrono::steady_clock 类的 typedef，或者它可以是单独的类型。

尽管 std::chrono::high_resolution_clock 具有所有库中提供的时钟的最高精度，std::chrono::high_resolution_clock::now()仍然会占用一定的时间量。在对短操作进行计时的时候你必须小心的评估对 std::chrono::high_resolution_clock::now()调用的开销。

类定义

```
class high_resolution_clock
{
public:
    typedef unspecified-integral-type rep;
    typedef std::ratio<
        unspecified,unspecified> period;
    typedef std::chrono::duration<rep,period> duration;
    typedef std::chrono::time_point<
        unspecified> time_point;
    static const bool is_steady=unspecified;

    static time_point now() noexcept;
};
```

# D.2 <condition_variable>头文件

<condition_variable>头文件提供了条件变量。这些都是基本级别的同步机制，可以允许一个线程阻塞直到某些条件为真或者经过了超时期限。

头文件内容

```
namespace std
{
    enum class cv_status { timeout, no_timeout };
    class condition_variable;
    class condition_variable_any;
}
```

## D.2.1 std::condition_variable 类

std::condition_variable 类允许线程等待一个条件变为 true。

std::condition_variable 的实例，不是 CopyAssignable、CopyConstructible、MoveAssignable 和 MoveConstructible 的。

类定义

```
class condition_variable
{
public:
    condition_variable();
    ~condition_variable();

    condition_variable(condition_variable const& ) = delete;
    condition_variable& operator=(condition_variable const& ) = delete;

    void notify_one() noexcept;
    void notify_all() noexcept;

    void wait(std::unique_lock<std::mutex>& lock);

    template <typename Predicate>
    void wait(std::unique_lock<std::mutex>& lock,Predicate pred);

    template <typename Clock, typename Duration>
    cv_status wait_until(
        std::unique_lock<std::mutex>& lock,
        const std::chrono::time_point<Clock, Duration>& absolute_time);

    template <typename Clock, typename Duration, typename Predicate>
    bool wait_until(
        std::unique_lock<std::mutex>& lock,
        const std::chrono::time_point<Clock, Duration>& absolute_time,
        Predicate pred);

    template <typename Rep, typename Period>
    cv_status wait_for(
        std::unique_lock<std::mutex>& lock,
        const std::chrono::duration<Rep, Period>& relative_time);

    template <typename Rep, typename Period, typename Predicate>
    bool wait_for(
        std::unique_lock<std::mutex>& lock,
        const std::chrono::duration<Rep, Period>& relative_time,
        Predicate pred);
};

void notify_all_at_thread_exit(condition_variable&,unique_lock<mutex>);
```

### std::condition_variable 默认构造函数

构造 std::condition_variable 对象。

声明

```
condition_variable();
```

结果

构造一个新的 std::condition_variable 实例。

引发

如果条件变量无法构造，抛出 std::system_error 类型的异常。

### std::condition_variable 析构函数

销毁 std::condition_variable 对象。

声明

```
~condition_variable();
```

前置条件

在对 wait()、wait_for()和 wait_until()的调用中，没有线程被阻塞在*this 上。

结果

销毁*this。

引发

无。

### std::condition_variable::notify_one 成员函数

唤醒当前在 std::condition_variable 上等待的其中一条线程。

声明

```
void notify_one() noexcept;
```

结果

在调用处，唤醒在*this 上等待的其中一条线程。如果没有线程等待着，此调用没有任何效果。

引发

如果无法得到结果，引发 std::system_error 异常。

同步

在单一 std::condition_variable 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::notify_all 成员函数

唤醒当前在 std::condition_variable 上等待的全部线程。

声明

```
void notify_all() noexcept;
```

**结果**

在调用处，唤醒在*this 上等待的所有线程。如果没有线程等待着，此调用没有任何效果。

**引发**

如果无法得到结果，引发 std::system_error 异常。

**同步**

在单一 std::condition_variable 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait 成员函数

一直等待，直到 std::condition_variable 通过调用 notify_one()、notify_all()或伪唤醒而被唤醒。

**声明**

```
void wait(std::unique_lock<std::mutex>& lock);
```

**前置条件**

lock.owns_lock()为真，且该锁为调用线程所拥有。

**结果**

原子级解锁所提供的 lock 对象并阻塞，直到该线程被另一个线程通过调用 notify_one()、notify_all()而唤醒，或是线程自己伪唤醒。在对 wait()的调用返回之前，lock 对象被再次锁定。

**引发**

如果无法得到结果，引发 std::system_error 异常。如果 lock 对象在调用 wait()之中被解锁，它会在退出时再次被锁定，即便函数经由异常而退出。

**注** 伪唤醒的意思是调用 wait()的线程可能在没有线程调用过 notify_one()或 notify_all()的情况下唤醒。因此建议如果可能的话，首选接受断言的 wait()重载版本。否则，建议在一个测试与条件变量关联的断言的循环中来调用 wait()。

**同步**

在单一 std::condition_variable 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait 成员函数之接受断言的重载版本

一直等待，直到 std::condition_variable 通过调用 notify_one()、

notify_all()而被唤醒，且断言为true。

声明

```
template<typename Predicate>
void wait(std::unique_lock<std::mutex>& lock,Predicate pred);
```

前置条件

表达式pred()必须有效，且其返回的可转换为bool.lock.owns_lock()的值必须为true，且该锁必须被调用wait()的线程所拥有。

结果

类似于

```
while(!pred())
{
    wait(lock);
}
```

引发

因调用pred所引发的所有异常，或者如果无法得到结果，引发std::system_error异常。

注 潜在的伪唤醒的意思是无法确定pred会被调用多少次。pred总是被lock锁定的互斥元调用，而且当（且仅当）(bool)pred()返回true时该函数才会返回。

同步

在单一 std::condition_variable 实例上对 notify_one()、notify_all()、wait()、wait_for()和wait_until()的调用会被序列化。对notify_one()或notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait_for成员函数

一直等待，直到std::condition_variable通过调用notify_one()、notify_all()或伪唤醒而被唤醒，或者直到一个指定的时间段逝去或线程被伪唤醒。

声明

```
template<typename Rep,typename Period>
cv_status wait_for(
    std::unique_lock<std::mutex>& lock,
    std::chrono::duration<Rep,Period> const& relative_time);
```

前置条件

lock.owns_lock()为真，且该锁为调用线程所拥有。

结果

原子级解锁所提供的lock对象并阻塞，直到该线程被另一个线程通过调用notify_one()、notify_all()而唤醒，或者relative_time指定的时间段逝去，

或是线程自己伪唤醒。在对wait_for()的调用返回之前，lock对象被再次锁定。

引发

如果无法得到结果，引发std::system_error异常。如果lock对象在调用wait()之中被解锁，它会在退出时再次被锁定，即便函数经由异常而退出。

注 伪唤醒的意思是调用wait_for()的线程可能在没有线程调用过notify_one()或notify_all()的情况下唤醒。因此建议如果可能的话，首选接受断言的wait_for()重载版本。否则，建议在一个测试与条件变量关联的断言的循环中来调用 wait_for()。当做此工作来确保超时仍然有效的时候必须注意，wait_until()在多数场合下可能会更合适。线程可能会比指定的时间段阻塞更久。如果可能，逝去的时间应决定于匀速时钟。

同步

在单一std::condition_variable实例上对notify_one()、notify_all()、wait()、wait_for()和wait_until()的调用会被序列化。对notify_one()或notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait_for 成员函数之接受断言的重载版本

一直等待，直到std::condition_variable通过调用notify_one()、notify_all()且断言为true，或者直到一个指定的时间段逝去。

声明

```
template<typename Rep,typename Period,typename Predicate>
bool wait_for(
    std::unique_lock<std::mutex>& lock,
    std::chrono::duration<Rep,Period> const& relative_time,
    Predicate pred);
```

前置条件

表达式pred()必须有效，且其返回的可转换为bool.lock.owns_lock()的值必须为true，且该锁必须被调用wait_for()的线程所拥有。

结果

类似于

```
internal_clock::time_point end=internal_clock::now()+relative_time;
while(!pred())
{
    std::chrono::duration<Rep,Period> remaining_time=
        end-internal_clock::now();
    if(wait_for(lock,remaining_time)==std::cv_status::timeout)
        return pred();
}
return true;
```

**返回**

如果对 pred()最近的调用返回 true，则返回 true，如果由 relative_time 指定的时间间隔逝去且 pred()返回 false，则返回 false。

注 潜在的伪唤醒的意思是无法确定 pred 会被调用多少次。pred 总是被 lock 锁定的互斥元调用，而且当（且仅当）(bool)pred()返回 true 或者 relative_time 指定的时间段逝去时该函数才会返回。线程可能会比指定的时间段阻塞更久。如果可能，逝去的时间应决定于匀速时钟。

**引发**

因调用 pred 所引发的所有异常，或者如果无法得到结果，引发 std::system_error 异常。

**同步**

在单一 std::condition_variable 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait_until 成员函数

一直等待，直到 std::condition_variable 通过调用 notify_one()、notify_all()或伪唤醒而被唤醒，或者达到一个指定的时间，或线程被伪唤醒。

**声明**

```
template<typename Clock,typename Duration>
cv_status wait_until(
    std::unique_lock<std::mutex>& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

**前置条件**

lock.owns_lock()为真，且该锁为调用线程所拥有。

**结果**

原子地解锁所提供的 lock 对象并阻塞，直到该线程被另一个线程通过调用 notify_one()、notify_all()而唤醒，或者 Clock::now()返回了一个等于或晚于 absolute_time 的时间，或是线程自己伪唤醒。在对 wait_until()的调用返回之前，lock 对象被再次锁定。

**返回**

如果线程被 notify_one()或 notify_all()的调用唤醒或者伪唤醒，返回 std::cv_status::no_timeout，否则返回 std::cv_status::timeout。

**引发**

如果无法得到结果，引发 std::system_error 异常。如果 lock 对象在调用

wait()之中被解锁，它会在退出时再次被锁定，即便函数经由异常而退出。

注 伪唤醒的意思是调用 wait_until()的线程可能在没有线程调用过 notify_one()或 notify_all()的情况下唤醒。因此建议如果可能的话，首选接受断言的 wait_until()重载版本。否则，建议在一个测试与条件变量关联的断言的循环中来调用 wait_until()。没有保证说调用线程会被阻塞多久，只有当函数返回 false，且 Clock::now()返回的时间等于或晚于 absolute_time 的时间点，线程才会解除阻塞。

同步

在单一 std::condition_variable 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait_until 成员函数之接受断言的重载版本

一直等待，直到 std::condition_variable 通过调用 notify_one()、notify_all()且断言为 true，或者达到直到一个指定的时间。

声明

```
template<typename Clock,typename Duration,typename Predicate>
bool wait_until(
    std::unique_lock<std::mutex>& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time,
    Predicate pred);
```

前置条件

表达式 pred()必须有效，且其返回的可转换为 bool.lock.owns_lock()的值必须为 true，且该锁必须被调用 wait_until()的线程所拥有。

结果

类似于

```
while(!pred())
{
    if(wait_until(lock,absolute_time)==std::cv_status::timeout)
        return pred();
}
return true;
```

返回

如果对 pred()最近的调用返回 true，则返回 true，如果由 relative_time 指定的时间间隔逝去且 pred()返回 false，则返回 false。

注 潜在的伪唤醒的意思是无法确定 pred 会被调用多少次。pred 总是被 lock 锁定的互斥元调用，而且当（且仅当）(bool)pred()返回 true 或者 Clock::now()返回一个等

于或晚于absolute_time的时间，函数才会返回。没有保证说调用线程会被阻塞多久，只有当函数返回false，且Clock::now()返回的时间等于或晚于absolute_time的时间点，线程才会解除阻塞。

引发

因调用 pred 所引发的所有异常，或者如果无法得到结果，引发std::system_error异常。

同步

在单一 std::condition_variable 实例上对 notify_one()、notify_all()、wait()、wait_for()和wait_until()的调用会被序列化。对notify_one()或notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::notify_all_at_thread_exit 非成员函数

在当前线程退出时，唤醒所有等待std::condition_variable的线程。

声明

```
void notify_all_at_thread_exit(
    condition_variable& cv,unique_lock<mutex> lk);
```

前置条件

lk.owns_lock()为true，并且该锁被调用线程持有。lk.mutex()应该返回相同的值，作为任意提供给wait()、wait_for()或wait_until()的锁对象，在来自当前等待线程的cv上。

结果

将lk持有的锁的所有权转移给内部存储，并且计划当调用线程退出时通知cv。此通知应该类似于：

```
lk.unlock();
cv.notify_all();
```

引发

如果无法达成该结果，引发std:system_error异常。

注 锁会一直持有到线程退出，所以必须小心避免死锁。建议调用线程应尽早退出，并且在该线程上不要进行阻塞操作。

用户应该确保等待线程不会在被唤醒的时候错误地假设线程已退出，尤其有潜在的伪唤醒时。要达到这一目的，可以在等待线程上测试一个只会做出true的断言，在互斥元的保护下通知线程，且不在调用notify_all_at_thread_exit之前释放互斥元上的锁。

## D.2.2 std::condition_variable_any 类

std::condition_variable 类允许线程等待一个条件变为 true。这里 std::condition_variable 只能与 std::unique_lock<std::mutex>一起使用，std::condition_variable_any 可以与任意符合 Lockable 需求的类型一起使用。

std::condition_variable_any 的实例，不是 CopyAssignable、CopyConstructible、MoveAssignable 和 MoveConstructible 的。

**类定义**

```
class condition_variable_any
{
public:
  condition_variable_any();
  ~condition_variable_any();

  condition_variable_any(
      condition_variable_any const& ) = delete;
  condition_variable_any& operator=(
      condition_variable_any const& ) = delete;

  void notify_one() noexcept;
  void notify_all() noexcept;

  template<typename Lockable>
  void wait(Lockable& lock);

  template <typename Lockable, typename Predicate>
  void wait(Lockable& lock, Predicate pred);

  template <typename Lockable, typename Clock,typename Duration>
  std::cv_status wait_until(
      Lockable& lock,
      const std::chrono::time_point<Clock, Duration>& absolute_time);

  template <
      typename Lockable, typename Clock,
      typename Duration, typename Predicate>
  bool wait_until(
      Lockable& lock,
      const std::chrono::time_point<Clock, Duration>& absolute_time,
      Predicate pred);

  template <typename Lockable, typename Rep, typename Period>
  std::cv_status wait_for(
      Lockable& lock,
      const std::chrono::duration<Rep, Period>& relative_time);
   template <
      typename Lockable, typename Rep,
      typename Period, typename Predicate>
   bool wait_for(
      Lockable& lock,
      const std::chrono::duration<Rep, Period>& relative_time,
      Predicate pred);
};
```

### std::condition_variable_any 默认构造函数

构造 std::condition_variable_any 对象。

**声明**

```
condition_variable_any();
```

**结果**

构造一个新的 std::condition_variable_any 实例。

**引发**

如果条件变量无法构造，引发 std::system_error 类型的异常。

### std::condition_variable_any 析构函数

销毁 std::condition_variable_any 对象。

**声明**

```
~condition_variable_any();
```

**前置条件**

在对 wait()、wait_for()和 wait_until()的调用中，没有线程被阻塞在*this 上。

**结果**

销毁*this。

**引发**

无。

### std::condition_variable_any::notify_one 成员函数

唤醒当前在 std::condition_variable_any 上等待的其中一条线程。

**声明**

```
void notify_one() noexcept;
```

**结果**

在调用处，唤醒在*this 上等待的其中以一条线程。如果没有线程等待着，此调用没有任何效果。

**引发**

如果无法得到结果，引发 std::system_error 异常。

**同步**

在单一 std::condition_variable_any 实例上对 notify_one()、

notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable_any::notify_all 成员函数

唤醒当前在 std::condition_variable_any 上等待的全部线程。

**声明**

```
void notify_all() noexcept;
```

**结果**

在调用处，唤醒在*this 上等待的所有线程。如果没有线程等待着，此调用没有任何效果。

**引发**

如果无法得到结果，引发 std::system_error 异常。

**同步**

在单一 std::condition_variable_any 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable_any::wait 成员函数

一直等待，直到 std::condition_variable_any 通过调用 notify_one()、notify_all()而被唤醒或伪唤醒。

**声明**

```
template<typename Lockable>
void wait(Lockable& lock);
```

**前置条件**

Lockable 满足 Lockable 需求，且 lock 拥有锁。

**结果**

原子级解锁所提供的 lock 对象并阻塞，直到该线程被另一个线程通过调用 notify_one()、notify_all()而唤醒，或是线程自己伪唤醒。在对 wait()的调用返回之前，lock 对象被再次锁定。

**引发**

如果无法得到结果，引发 std::system_error 异常。如果 lock 对象在调用 wait()之中被解锁，它会在退出时再次被锁定，即便函数经由异常而退出。

**注** 伪唤醒的意思是调用 wait()的线程可能在没有线程调用过 notify_one()或 notify_all()的情况下唤醒。因此建议如果可能的话，首选接受断言的 wait()重载版

本。否则，建议在一个测试与条件变量关联的断言的循环中来调用 wait()。

**同步**

在单一 std::condition_variable_any 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable_any::wait 成员函数之接受断言的重载版本

一直等待，直到 std::condition_variable_any 通过调用 notify_one()、notify_all()而被唤醒，且断言为 true。

**声明**

```
template<typename Lockable,typename Predicate>
void wait(Lockable& lock,Predicate pred);
```

**前置条件**

表达式 pred()必须有效，且其返回的可转换为 bool.lock.owns_lock()的值必须为 true。Lockable 满足 Lockable 需求，且 lock 拥有锁。

**结果**

类似于

```
while(!pred())
{
wait(lock);
}
```

**引发**

因调用 pred 所引发的所有异常，或者如果无法得到结果，引发 std::system error 异常。

注 潜在的伪唤醒的意思是无法确定 pred 会被调用多少次。pred 总是被 lock 锁定的互斥元调用，而且当（且仅当）(bool)pred()返回 true 时该函数才会返回。

**同步**

在单一 std::condition_variable_any 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable_any::wait_for 成员函数

一直等待，直到 std::condition_variable 通过调用 notify_one()、notify_all()或伪唤醒而被唤醒，或者直到一个指定的时间段逝去或线程被伪唤醒。

**声明**

```
template<typename Lockable,typename Rep,typename Period>
std::cv_status wait_for(
    Lockable& lock,
    std::chrono::duration<Rep,Period> const& relative_time);
```

**前置条件**

Lockable 满足 Lockable 需求，且 lock 拥有锁。

**结果**

原子级解锁所提供的 lock 对象并阻塞，直到该线程被另一个线程通过调用 notify_one()、notify_all()而唤醒，或者 relative_time 指定的时间段逝去，或是线程自己伪唤醒。在对 wait_for()的调用返回之前，lock 对象被再次锁定。

**引发**

如果无法得到结果，引发 std::system_error 异常。如果 lock 对象在调用 wait()之中被解锁，它会在退出时再次被锁定，即便函数经由异常而退出。

**注** 伪唤醒的意思是调用 wait_for()的线程可能在没有线程调用过 notify_one()或 notify_all()的情况下唤醒。因此建议如果可能的话，首选接受断言的 wait_for()重载版本。否则，建议在一个测试与条件变量关联的断言的循环中来调用 wait_for()。当做此工作来确保超时仍然有效的时候必须注意；wait_until()在多数场合下可能会更合适。线程可能会比指定的时间段阻塞更久。如果可能，逝去的时间应决定于匀速时钟。

**同步**

在单一 std::condition_variable_any 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable_any::wait_for 成员函数之接受断言的重载版本

一直等待，直到 std::condition_variable_any 通过调用 notify_one()、notify_all()且断言为 true，或者直到一个指定的时间段逝去。

**声明**

```
template<typename Lockable,typename Rep,
    typename Period, typename Predicate>
bool wait_for(
    Lockable& lock,
    std::chrono::duration<Rep,Period> const& relative_time,
    Predicate pred);
```

**前置条件**

表达式 pred()必须有效，且其返回的可转换为 bool.lock。Lockable 满足

Lockable 需求，且 lock 拥有锁。

**结果**

类似于

```
internal_clock::time_point end=internal_clock::now()+relative_time;
while(!pred())
{
    std::chrono::duration<Rep,Period> remaining_time=
        end-internal_clock::now();
    if(wait_for(lock,remaining_time)==std::cv_status::timeout)
        return pred();
}
return true;
```

**返回**

如果对 pred()最近的调用返回 true，则返回 true，如果由 relative_time 指定的时间间隔逝去且 pred()返回 false，则返回 false。

注　潜在的伪唤醒的意思是无法确定 pred 会被调用多少次。pred 总是被 lock 锁定的互斥元调用，而且当（且仅当）(bool)pred()返回 true 或者 relative_time 指定的时间段逝去时该函数才会返回。线程可能会比指定的时间段阻塞更久。如果可能，逝去的时间应决定于匀速时钟。

**引发**

因调用 pred 所引发的所有异常，或者如果无法得到结果，引发 std::system_error 异常。

**同步**

在单一 std::condition_variable_any 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait_until 成员函数

一直等待，直到 std::condition_variable 通过调用 notify_one()、notify_all()或伪唤醒而被唤醒，或者达到一个指定的时间，或线程被伪唤醒。

**声明**

```
template<typename Lockable,typename Clock,typename Duration>
std::cv_status wait_until(
    Lockable& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

**前置条件**

lock.owns_lock()为真，且该锁为调用线程所拥有。

**结果**

原子级解锁所提供的 lock 对象并阻塞，直到该线程被另一个线程通过调用 notify_one()、notify_all()而唤醒，或者 Clock::now()返回了一个等于或晚于 absolute_time 的时间，或是线程自己伪唤醒。在对 wait_until()的调用返回之前，lock 对象被再次锁定。

**返回**

如果线程被 notify_one()或 notify_all()的调用唤醒或者伪唤醒，返回 std::cv_status::no_timeout，否则返回 std::cv_status::timeout。

**引发**

如果无法得到结果，引发 std::system_error 异常。如果 lock 对象在调用 wait()之中被解锁，它会在退出时再次被锁定，即便函数经由异常而退出。

**注** 伪唤醒的意思是调用 wait()的线程可能在没有线程调用过 notify_one()或 notify_all()的情况下唤醒。因此建议如果可能的话，首选接受断言的 wait()重载版本。否则，建议在一个测试与条件变量关联的断言的循环中来调用 wait_until()。没有保证说调用线程会被阻塞多久，只有当函数返回 false，且 Clock::now()返回的时间等于或晚于 absolute_time 的时间点，线程才会解除阻塞。

**同步**

在单一 std::condition_variable_any 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

### std::condition_variable::wait_until 成员函数之接受断言的重载版本

一直等待，直到 std::condition_variable 通过调用 notify_one()、notify_all()且断言为 true，或者直到达到一个指定的时间。

**声明**

```
template<typename Lockable,typename Clock,
    typename Duration, typename Predicate>
bool wait_until(
    Lockable& lock,
    std::chrono::time_point<Clock,Duration> const& absolute_time,
    Predicate pred);
```

**前置条件**

表达式 pred()必须有效，且其返回的可转换为 bool.lock.owns_lock()的值必须为 true，且该锁必须被调用 wait()的线程所拥有。

结果

类似于

```
while(!pred())
{
    if(wait_until(lock,absolute_time)==std::cv_status::timeout)
        return pred();
}
return true;
```

返回

如果对 pred()最近的调用返回 true，则返回 true，如果由 relative_time 指定的时间间隔逝去且 pred()返回 false，则返回 false。

注　潜在的伪唤醒的意思是无法确定 pred 会被调用多少次。pred 总是被 lock 锁定的互斥元调用，而且当（且仅当）(bool)pred()返回 true 或者 Clock::now()返回一个等于或晚于 absolute_time 的时间，函数才会返回。没有保证说调用线程会被阻塞多久，只有当函数返回 false，且 Clock::now()返回的时间等于或晚于 absolute_time 的时间点，线程才会解除阻塞。

引发

因调用 pred 所引发的所有异常，或者如果无法得到结果，引发 std::system_error 异常。

同步

在单一 std::condition_variable_any 实例上对 notify_one()、notify_all()、wait()、wait_for()和 wait_until()的调用会被序列化。对 notify_one()或 notify_all()的调用，只会唤醒在该调用之前就开始等待的线程。

## D.3　<atomic>头文件

<atomic>头文件提供了一组基本的原子类型以及对这些类型的操作，还有一个类模板，用来构造满足一些条件的用户定义类型的原子版本。

头文件内容

```
#define ATOMIC_BOOL_LOCK_FREE see description
#define ATOMIC_CHAR_LOCK_FREE see description
#define ATOMIC_SHORT_LOCK_FREE see description
#define ATOMIC_INT_LOCK_FREE see description
#define ATOMIC_LONG_LOCK_FREE see description
#define ATOMIC_LLONG_LOCK_FREE see description
#define ATOMIC_CHAR16_T_LOCK_FREE see description
#define ATOMIC_CHAR32_T_LOCK_FREE see description
#define ATOMIC_WCHAR_T_LOCK_FREE see description
```

```
#define ATOMIC_POINTER_LOCK_FREE see description

#define ATOMIC_VAR_INIT(value) see description

namespace std
{
    enum memory_order;

    struct atomic_flag;
    typedef see description atomic_bool;
    typedef see description atomic_char;
    typedef see description atomic_char16_t;
    typedef see description atomic_char32_t;
    typedef see description atomic_schar;
    typedef see description atomic_uchar;
    typedef see description atomic_short;
    typedef see description atomic_ushort;
    typedef see description atomic_int;
    typedef see description atomic_uint;
    typedef see description atomic_long;
    typedef see description atomic_ulong;
    typedef see description atomic_llong;
    typedef see description atomic_ullong;
    typedef see description atomic_wchar_t;

    typedef see description atomic_int_least8_t;
    typedef see description atomic_uint_least8_t;
    typedef see description atomic_int_least16_t;
    typedef see description atomic_uint_least16_t;
    typedef see description atomic_int_least32_t;
    typedef see description atomic_uint_least32_t;
    typedef see description atomic_int_least64_t;
    typedef see description atomic_uint_least64_t;
    typedef see description atomic_int_fast8_t;
    typedef see description atomic_uint_fast8_t;
    typedef see description atomic_int_fast16_t;
    typedef see description atomic_uint_fast16_t;
    typedef see description atomic_int_fast32_t;
    typedef see description atomic_uint_fast32_t;
    typedef see description atomic_int_fast64_t;
    typedef see description atomic_uint_fast64_t;
    typedef see description atomic_int8_t;
    typedef see description atomic_uint8_t;
    typedef see description atomic_int16_t;
    typedef see description atomic_uint16_t;
    typedef see description atomic_int32_t;
    typedef see description atomic_uint32_t;
    typedef see description atomic_int64_t;
    typedef see description atomic_uint64_t;
    typedef see description atomic_intptr_t;
    typedef see description atomic_uintptr_t;
    typedef see description atomic_size_t;
    typedef see description atomic_ssize_t;
    typedef see description atomic_ptrdiff_t;
    typedef see description atomic_intmax_t;
```

```
    typedef see description atomic_uintmax_t;

    template<typename T>
    struct atomic;

    extern "C" void atomic_thread_fence(memory_order order);
    extern "C" void atomic_signal_fence(memory_order order);

    template<typename T>
    T kill_dependency(T);
}
```

## D.3.1 std::atomic_xxx typedef

为了向下兼容C标准，提供了原子整型的typedef。这既是对相应 std::atomic<T> 特化的typedef，也是具有相同接口特化的基类。

| std::atomic_*itype* | std::atomic<>特化 |
|---|---|
| std::atomic_char | std::atomic<char> |
| std::atomic_schar | std::atomic<signed char> |
| std::atomic_uchar | std::atomic<unsigned char> |
| std::atomic_short | std::atomic<short> |
| std::atomic_ushort | std::atomic<unsigned short> |
| std::atomic_int | std::atomic<int> |
| std::atomic_uint | std::atomic<unsigned int> |
| std::atomic_long | std::atomic<long> |
| std::atomic_ulong | std::atomic<unsigned long> |
| std::atomic_llong | std::atomic<long long> |
| std::atomic_ullong | std::atomic<unsigned long long> |
| std::atomic_wchar_t | std::atomic<wchar_t> |
| std::atomic_char16_t | std::atomic<char16_t> |
| std::atomic_char32_t | std::atomic<char32_t> |

## D.3.2 ATOMIC_xxx_LOCK_FREE 宏

这些宏确定了对应特定内置类型的源自类型是不是无锁的。

**宏声明**

```
#define ATOMIC_BOOL_LOCK_FREE see description
#define ATOMIC_CHAR_LOCK_FREE see description
#define ATOMIC_SHORT_LOCK_FREE see description
#define ATOMIC_INT_LOCK_FREE see description
#define ATOMIC_LONG_LOCK_FREE see description
#define ATOMIC_LLONG_LOCK_FREE see description
#define ATOMIC_CHAR16_T_LOCK_FREE see description
#define ATOMIC_CHAR32_T_LOCK_FREE see description
#define ATOMIC_WCHAR_T_LOCK_FREE see description
#define ATOMIC_POINTER_LOCK_FREE see description
```

ATOMIC_xxx_LOCK_FREE 的值是 0、1 或 2。值 0 表示该操作对于提名类型对应的有符号和无符号原子类型从来都不是无锁的，值 1 表示该操作对于那些类型在特定场合下可能是无锁的，而其他场合则不是，值 2 表示该操作始终是无锁的。例如，如果 ATOMIC_INT_LOCK_FREE 是 2，std::atomic<int>和 std::atomic<unsigned>上的操作始终是无锁的。

ATOMIC_POINTER_LOCK_FREE 宏描述了在原子指针特化 std::atomic<T*>上的操作的无锁属性。

## D.3.3 ATOMIC_VAR_INIT 宏

ATOMIC_VAR_INIT 宏提供了将原子变量初始化至特定值的方法。

**声明**

```
#define ATOMIC_VAR_INIT(value) see description
```

这个宏展开至令牌序列，能够以下面的形式，用来在表达式里使用指定值初始化标准原子类型：

```
std::atomic<type> x = ATOMIC_VAR_INIT(val);
```

指定的值必须与对应于原子类型的非原子类型相兼容，例如：

```
std::atomic<int> i = ATOMIC_VAR_INIT(42);
std::string s;
std::atomic<std::string*> p = ATOMIC_VAR_INIT(&s);
```

这样的初始化不是原子的，并且任何通过其他线程访问即将初始化的变量时，初始化没有发生于访问之前就会产生数据竞争而且是未定义的行为。

## D.3.4 std::memory_order 枚举

std::memory_order 枚举用来指定原子操作的排序约束。

**声明**

```
typedef enum memory_order
{
    memory_order_relaxed,memory_order_consume,
    memory_order_acquire,memory_order_release,
    memory_order_acq_rel,memory_order_seq_cst
} memory_order;
```

标记有这些内存顺序值的操作表现如下（参见第 5 章）。

### std::memory_order_relaxed

此操作不会提供任何额外的排序约束。

std::memory_order_release

此操作是在指定内存地点的释放操作。它因此在与一个读取存储值相同内存地点的获取操作同步。

std::memory_order_acquire

此操作是在指定内存地点的获取操作。如果存储的值被一个释放操作所写，则该存储与此操作同步。

std::memory_order_acq_rel

此操作必须是读-修改-写操作，并且在指定地点同时表现为 std::memory_order_acquire 和 std::memory_order_release。

std::memory_order_seq_cst

此操作构成顺序一致操作的一个全局总体排序的一部分。另外，如果这是个存储，它表现为 std::memory_order_release 操作，如果这是个载入，它表现为 std::memory_order_acquire 操作，如果它是读-修改-写操作，它同时表现为 std::memory_order_acquire 和 std::memory_order_release。这是所有操作的默认值。

std::memory_order_consume

此操作是在指定的内存位置的消费操作。

## D.3.5 std::atomic_thread_fence 函数

std::atomic_thread_fence()在代码中插入一个“内存障碍”或是“屏障”，以便在操作之间强制内存顺序约束。

声明

```
extern "C" void atomic_thread_fence(std::memory_order order);
```

结果

插入一个带有所需内存顺序约束的屏障。

带有 std::memory_order_release、std::memory_order_acq_rel 或 std::memory_order_seq_cst 的 order 的屏障，与相同内存地址上的获取操作同步，如果该获取操作读取一个由屏障后面原子操作存储的值在相同的线程上。

释放操作与带有 std::memory_order_acquire、std::memory_order_

acq_rel 或 std::memory_order_seq_cst 的 order 的屏障同步，如果该释放操作存储一个被屏障前的原子操作读取的值。

引发

无。

### D.3.6 std::atomic_signal_fence 函数

std::atomic_signal_fence() 函数在代码中插入一个内存障碍或屏障，以便在线程上的操作和线程上信号句柄中的操作之间强制内存顺序约束

声明

```
extern "C" void atomic_signal_fence(std::memory_order order);
```

结果

插入一个带有所需内存顺序约束的屏障。等效于 std::atomic_thread_fence(order)，除了此约束只应用在线程和同一线程上的信号句柄之间。

引发

无。

### D.3.7 std::atomic_flag 类

std::atomic_flag 类提供了原子 flag 的简单骨架。这是 C++11 标准唯一保证为无锁的数据类型（尽管很多原子类型在大多数实现中也是无锁的）。

std::atomic_flag 的实例要么是 **set** 要么是 **clear**。

类定义

```
struct atomic_flag
{
    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;

    bool test_and_set(memory_order = memory_order_seq_cst) volatile
     noexcept;
    bool test_and_set(memory_order = memory_order_seq_cst) noexcept;
    void clear(memory_order = memory_order_seq_cst) volatile noexcept
    void clear(memory_order = memory_order_seq_cst) noexcept;
};

bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
bool atomic_flag_test_and_set(atomic_flag*) noexcept;
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag*, memory_order) noexcept;
```

```
bool atomic_flag_test_and_set_explicit(
    atomic_flag*, memory_order) noexcept;
void atomic_flag_clear(volatile atomic_flag*) noexcept;
void atomic_flag_clear(atomic_flag*) noexcept;
void atomic_flag_clear_explicit(
    volatile atomic_flag*, memory_order) noexcept;
void atomic_flag_clear_explicit(
    atomic_flag*, memory_order) noexcept;
#define ATOMIC_FLAG_INIT unspecified
```

### std::atomic_flag 默认构造函数

默认构造的 std::atomic_flag 实例是 clear 还是 set 是未指定的。对于静态存储时限的对象，初始化应该是静态初始化。

声明

```
std::atomic_flag() noexcept = default;
```

结果

构造在未指定的状态的新的 std::atomic_flag 对象。

引发

无。

### std::atomic_flag 使用 atomic_flag_init 初始化

std::atomic_flag 的实例可以使用 ATOMIC_FLAG_INIT 宏来初始化，这种情况下它会初始化为 **clear** 状态。对于静态存储时限的对象，初始化应该是静态初始化。

声明

```
#define ATOMIC_FLAG_INIT unspecified
```

用法

```
std::atomic_flag flag=ATOMIC_FLAG_INIT;
```

结果

构造在 **clear** 状态的新的 std::atomic_flag 对象。

引发

无。

### std::atomic_flag::test_and_set 成员函数

原子级设置 flag，并检查它是不是已设置。

声明

```
bool test_and_set(memory_order order = memory_order_seq_cst) volatile
    noexcept;
bool test_and_set(memory_order order = memory_order_seq_cst) noexcept;
```

结果

设置 flag。

返回

如果 flag 在调用处是已设置的，返回 `true`，如果 flag 是清除的，返回 `false`。

引发

无。

**注** 这是一项对于包括*this 的内存地址的原子的读-修改-写操作。

### std::atomic_flag_test_and_set 非成员函数

设置 flag，并检查它是不是已设置。

声明

```
bool atomic_flag_test_and_set(volatile atomic_flag* flag) noexcept;
bool atomic_flag_test_and_set(atomic_flag* flag) noexcept;
```

结果

```
return flag->test_and_set();
```

### std::atomic_flag_test_and_set_explicit 非成员函数

设置 flag，并检查它是不是已设置。

声明

```
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag* flag, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(
    atomic_flag* flag, memory_order order) noexcept;
```

返回

```
return flag->test_and_set(order);
```

### std::atomic_flag::clear 成员函数

清除 flag。

声明

```
void clear(memory_order order = memory_order_seq_cst) volatile noexcept;
void clear(memory_order order = memory_order_seq_cst) noexcept;
```

**前置条件**

提供的 order 必须是 std::memory_order_relaxed、std::memory_order_release 或 std::memory_order_seq_cst 其中之一。

**结果**

清除 flag。

**引发**

无。

注 这是对包括*this 的内存地址的原子存储操作。

### std::atomic_flag_clear 非成员函数

清除 flag。

**声明**

```
void atomic_flag_clear(volatile atomic_flag* flag) noexcept;
void atomic_flag_clear(atomic_flag* flag) noexcept;
```

**结果**

```
flag->clear();
```

### std::atomic_flag_clear_explicit 非成员函数

清除 flag。

**声明**

```
void atomic_flag_clear_explicit(
    volatile atomic_flag* flag, memory_order order) noexcept;
void atomic_flag_clear_explicit(
    atomic_flag* flag, memory_order order) noexcept;
```

**结果**

```
return flag->clear(order);
```

## D.3.8 std::atomic 类模板

std::atomic 类模板提供了一个带有原子操作的封装器，可以用于任意符合下面需求的类型。

模板参数 BaseType 必须具备如下特点。

- 有平凡的默认构造函数。
- 有平凡的拷贝赋值运算符。

■ 有平凡的析构函数。

■ 可以进行按位相等比较。

这基本上意味着 std::atomic<some-built-in-type>是正确的，正如 std::atomic<some-simple-struct>,但像 std::atomic<std::string>这样的就不行了。

在主模板之外，还有为内置整型而设的特化，和提供类似 x++这样额外操作的指针。

std::atomic 的实例不是 CopyConstructible 和 CopyAssignable 的，因为这些操作都不能作为一个单一原子操作来进行。

类定义

```
template<typename BaseType>
struct atomic
{
    atomic() noexcept = default;
    constexpr atomic(BaseType) noexcept;
    BaseType operator=(BaseType) volatile noexcept;
    BaseType operator=(BaseType) noexcept;

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(BaseType,memory_order = memory_order_seq_cst)
        volatile noexcept;
    void store(BaseType,memory_order = memory_order_seq_cst) noexcept;
    BaseType load(memory_order = memory_order_seq_cst)
        const volatile noexcept;
    BaseType load(memory_order = memory_order_seq_cst) const noexcept;
    BaseType exchange(BaseType,memory_order = memory_order_seq_cst)
        volatile noexcept;
    BaseType exchange(BaseType,memory_order = memory_order_seq_cst)
        noexcept;

    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) volatile noexcept;
    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
```

```
        memory_order order = memory_order_seq_cst)
        volatile noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) volatile noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) noexcept;

    operator BaseType () const volatile noexcept;
    operator BaseType () const noexcept;
};

template<typename BaseType>
bool atomic_is_lock_free(volatile const atomic<BaseType>*) noexcept;
template<typename BaseType>
bool atomic_is_lock_free(const atomic<BaseType>*) noexcept;
template<typename BaseType>
void atomic_init(volatile atomic<BaseType>*, void*) noexcept;
template<typename BaseType>
void atomic_init(atomic<BaseType>*, void*) noexcept;
template<typename BaseType>
BaseType atomic_exchange(volatile atomic<BaseType>*, memory_order)
    noexcept;
template<typename BaseType>
BaseType atomic_exchange(atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
    volatile atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
    atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
void atomic_store(volatile atomic<BaseType>*, BaseType) noexcept;
template<typename BaseType>
void atomic_store(atomic<BaseType>*, BaseType) noexcept;
template<typename BaseType>
void atomic_store_explicit(
    volatile atomic<BaseType>*, BaseType, memory_order) noexcept;
template<typename BaseType>
void atomic_store_explicit(
    atomic<BaseType>*, BaseType, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>*) noexcept;
template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>*) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    volatile const atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
```

```
    const atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    volatile atomic<BaseType>*,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    atomic<BaseType>*,BaseType * old_value,BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

注　尽管这些非成员函数被指定为模板，它们可以作为函数的重载集来提供，且不应该使用模板参数的显式特化。

### std::atomic 默认构造函数

使用默认初始化值构造 std::atomic 的实例。

**声明**

```
atomic() noexcept;
```

**结果**

使用默认初始化值构造一个新的 std::atomic 的实例。对于静态存储时限的对象，初始化应该是静态初始化。

注 使用默认构造函数初始化的带有非静态存储时限的 std::atomic 实例，不能指望它拥有一个可预见的值。

**引发**

无。

## std::atomic_int 非成员函数

在 std::atomic<BaseType>的实例中非原子级存储给定的值。

**声明**

```
template<typename BaseType>
void atomic_init(atomic<BaseType> volatile* p, BaseType v) noexcept;
template<typename BaseType>
void atomic_init(atomic<BaseType>* p, BaseType v) noexcept;
```

**结果**

非原子级将 v 的值存储在*p 中。在一个尚未默认构造的或是在构造后已进行过任何操作的 atomic<BaseType>实例上调用 atomic_init()，都是未定义的行为。

注 由于该存储是非原子的，所有来自于另一线程的对 p 所指向的对象的并发访问（即便是原子的）均构成数据竞争。

**引发**

无。

## std::atomic 转换构造函数

用给定的 BaseType 值构造 std::atomic 实例。

**声明**

```
constexpr atomic(BaseType b) noexcept;
```

**结果**

用 b 的值构造新的 std::atomic 对象。对于静态存储时限的对象这是静态初始化。

**引发**

无。

## std::atomic 转换赋值运算符

在*this 中存储一个新的值。

**声明**

```
BaseType operator=(BaseType b) volatile noexcept;
BaseType operator=(BaseType b) noexcept;
```

结果

```
return this->store(b);
```

### std::atomic::is_lock_free 成员函数

确定在*this 上的操作是无锁的。

声明

```
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
```

返回

如果在*this 上的操作是无锁的，返回 true，否则 false。

引发

无。

### std::atomic::is_lock_free 非成员函数

确定在*this 上的操作是无锁的。

声明

```
template<typename BaseType>
bool atomic_is_lock_free(volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
bool atomic_is_lock_free(const atomic<BaseType>* p) noexcept;
```

结果

```
return p->is_lock_free();
```

### std::atomic::load 成员函数

原子级载入 std::atomic 实例的当前值。

声明

```
BaseType load(memory_order order = memory_order_seq_cst)
    const volatile noexcept;
BaseType load(memory_order order = memory_order_seq_cst) const noexcept;
```

前置条件

提供的 order 必须是 std::memory_order_relaxed、std::memory_order_release 或 std::memory_order_seq_cst 其中之一。

结果

原子级载入存储在*this 中的值。

返回

在调用时*this中存储的值。

引发

无。

注 这是对包括*this的内存地址的原子载入操作。

## std::atomic_load 非成员函数

原子级载入std::atomic实例的当前值。

声明

```
template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>* p) noexcept;
```

结果

```
return p->load();
```

## std::atomic_load 非成员函数

原子级载入std::atomic实例的当前值。

声明

```
template<typename BaseType>
BaseType atomic_load_explicit(
    volatile const atomic<BaseType>* p, memory_order order) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    const atomic<BaseType>* p, memory_order order) noexcept;
```

结果

```
return p->load(order);
```

## std::atomic::operator 基类型转换运算符

载入存储在*this中的值。

声明

```
operator BaseType() const volatile noexcept;
operator BaseType() const noexcept;
```

结果

```
return this->load();
```

### std::atomic::store 成员函数

在 atomic<BaseType>实例中存储一个新值。

**声明**

```
void store(BaseType new_value,memory_order order = memory_order_seq_cst)
    volatile noexcept;
void store(BaseType new_value,memory_order order = memory_order_seq_cst)
    noexcept;
```

**前置条件**

提供的 order 必须是 std::memory_order_relaxed、std::memory_order_release 或 std::memory_order_seq_cst 其中之一。

**结果**

在*this 中存储 new_value。

**引发**

无。

注 这是对包括*this 的内存地址的原子存储操作。

### std::atomic_store 非成员函数

在 atomic<BaseType>实例中存储一个新值。

**声明**

```
template<typename BaseType>
void atomic_store(volatile atomic<BaseType>* p, BaseType new_value)
    noexcept;
template<typename BaseType>
void atomic_store(atomic<BaseType>* p, BaseType new_value) noexcept;
```

**结果**

```
p->store(new_value);
```

### std::atomic_store_explicit 非成员函数

在 atomic<BaseType>实例中存储一个新值。

**声明**

```
template<typename BaseType>
void atomic_store_explicit(
    volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
    noexcept;
template<typename BaseType>
void atomic_store_explicit(
    atomic<BaseType>* p, BaseType new_value, memory_order order) noexcept;
```

结果

```
p->store(new_value,order);
```

## std::atomic::exchange 成员函数

存储一个新值并读取旧值。

声明

```
BaseType exchange(
    BaseType new_value,
    memory_order order = memory_order_seq_cst)
    volatile noexcept;
```

结果

在*this 中存储 new_value，并且获取现存的*this 值。

返回

在刚刚存储之前的*this 值。

引发

无。

注　这是一项对于包括*this 的内存地址的原子的读-修改-写操作。

## std::atomic_exchange 非成员函数

在 atomic<BaseType>实例中存储一个新的值并且读取之前的值。

声明

```
template<typename BaseType>
BaseType atomic_exchange(volatile atomic<BaseType>* p, BaseType new_value)
    noexcept;
template<typename BaseType>
BaseType atomic_exchange(atomic<BaseType>* p, BaseType new_value) noexcept;
```

结果

```
return p->exchange(new_value);
```

## std::atomic_exchange_explicit 非成员函数

在 atomic<BaseType>实例中存储一个新的值并且读取之前的值。

声明

```
template<typename BaseType>
BaseType atomic_exchange_explicit(
    volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
    noexcept;
```

```
template<typename BaseType>
BaseType atomic_exchange_explicit(
    atomic<BaseType>* p, BaseType new_value, memory_order order) noexcept;
```

**结果**

```
return p->exchange(new_value,order);
```

## std::atomic::compare_exchange_strong 成员函数

原子级将值与一个期望值进行比较，并且如果两个值相等，就存储新的值。如果两个值不相等，就用读取到的值更新期望值。

**声明**

```
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order)
    volatile noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order) noexcept;
```

**前置条件**

failure_order 不能是 std::memory_order_release 或 std::memory_order_acq_rel。

**结果**

将 expected 与*this 中存储的值进行按位比较，并且当相等时将 new_value 存储在*this 中，否则，将 expected 更新为读取到的值。

**返回**

如果*this 中存在的值与 expected 相等，返回 true，否则 false。

**引发**

无。

**注** 三参数的重载与带有 success_order==order 和 failure_order==order 的四参数重载是等效的，除非 order 是 std::memory_order_acq_rel 而 failure_order 是 std::memory_order_acquire，或者 order 是 std::memory_order_release 而 failure_order 是 std::memory_order_relaxed。

**注** 如果结果为 true，这就是对包括*this 的内存地址的原子读-修改-写操作，带有内存顺序 success_order；否则，它就是对包括*this 的内存地址的载入操作，带有内存顺

序 failure_order。

### std::atomic_compare_exchange_strong 非成员函数

将值与一个期望值进行比较，并且如果两个值相等，就存储新的值。如果两个值不相等，就用读取到的值更新期望值。

**声明**

```
template<typename BaseType>
bool atomic_compare_exchange_strong(
    volatile atomic<BaseType>* p,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    atomic<BaseType>* p,BaseType * old_value,BaseType new_value) noexcept;
```

**结果**

```
return p->compare_exchange_strong(*old_value,new_value);
```

### std::atomic::compare_exchange_weak 成员函数

将值与一个期望值进行比较，并且如果两个值相等且更新可以在原子级完成，就存储新的值。如果两个值不相等或是更新不能在原子级完成，就用读取到的值更新期望值。

**声明**

```
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) volatile noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order)
    volatile noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order) noexcept;
```

**前置条件**

failure_order 不能是 std::memory_order_release 或 std::memory_order_acq_rel。

**结果**

将 expected 与*this 中存储的值进行按位比较，并且当相等时将 new_value 存储在*this 中。如果两个值不相等或是更新不能原子级进行，就将 expected 更新为读取到的值。

**返回**

如果*this 中存在的值与 expected 相等且 new_value 成功存储在*this 中，返

回 true，否则 false。

引发

无。

注　三参数的重载与带有 success_order==order 和 failure_order==order 的四参数重载是等效的，除非 order 是 std::memory_order_acq_rel 而 failure_order 是 std::memory_order_acquire，或者 order 是 std::memory_order_release 而 failure_order 是 std::memory_order_relaxed。

注　如果结果为 true，这就是对包括*this 的内存地址的原子读-修改-写操作，带有内存顺序 success_order；否则，它就是对包括*this 的内存地址的载入操作，带有内存顺序 failure_order。

### std::atomic_compare_exchange_weak 非成员函数

将值与一个期望值进行比较，并且如果两个值相等，就存储新的值。如果两个值不相等，就用读取到的值更新期望值。

声明

```
template<typename BaseType>
bool atomic_compare_exchange_weak(
    volatile atomic<BaseType>* p,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    atomic<BaseType>* p,BaseType * old_value,BaseType new_value) noexcept;
```

结果

```
return p->compare_exchange_weak(*old_value,new_value);
```

### std::atomic_compare_exchange_weak 非成员函数

将值与一个期望值进行比较，并且如果两个值相等，就存储新的值。如果两个值不相等，就用读取到的值更新期望值。

声明

```
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

结果

```
return p->compare_exchange_weak(
*old_value,new_value,success_order,failure_order);
```

### D.3.9　std::atomic 模板的特化

std::atomic 类模板的特化提供给整型和指针类型。对于整型，这些特化在主模板提供的操作之外，又额外提供了原子级的加、减和按位操作。对于指针类型，这一特化在主模板提供的操作之外又额外提供了原子级的指针算数运算。

为下面的整型提供了特化：

```
std::atomic<bool>
std::atomic<char>
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

以及对所有类型 T 的 std::atomic<T*>。

### D.3.10　std::atomic<integral-type>特化

std::atomic 类模板的 std::atomic<integral-type>特化为每一个基本的整型提供了原子整型数据类型，同时带有一整套的操作。

下面的描述应用于这些 std::atomic<>类模板的特化。

```
std::atomic<char>
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

这些特化的实例不是 CopyConstructible 和 CopyAssignable 的，因为这些

操作都不能作为一个单一原子操作来进行。

类定义

```
template<>
struct atomic<integral-type>
{
    atomic() noexcept = default;
    constexpr atomic(integral-type) noexcept;
    bool operator=(integral-type) volatile noexcept;

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    void store(integral-type,memory_order = memory_order_seq_cst) noexcept;
    integral-type load(memory_order = memory_order_seq_cst)
        const volatile noexcept;
    integral-type load(memory_order = memory_order_seq_cst) const noexcept;
    integral-type exchange(
        integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    integral-type exchange(
        integral-type,memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_strong(
        integral-type & old_value,integral-type new_value,
         memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        integral-type & old_value,integral-type new_value,
         memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        integral-type & old_value,integral-type new_value,
        memory_order success_order,memory_order failure_order)
        volatile noexcept;
    bool compare_exchange_strong(
        integral-type & old_value,integral-type new_value,
        memory_order success_order,memory_order failure_order) noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type new_value,
         memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type new_value,
         memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type new_value,
        memory_order success_order,memory_order failure_order)
        volatile noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type new_value,
        memory_order success_order,memory_order failure_order) noexcept;
```

```
    operator integral-type() const volatile noexcept;
    operator integral-type() const noexcept;
    integral-type fetch_add(
        integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    integral-type fetch_add(
        integral-type,memory_order = memory_order_seq_cst) noexcept;
    integral-type fetch_sub(
        integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    integral-type fetch_sub(
        integral-type,memory_order = memory_order_seq_cst) noexcept;
    integral-type fetch_and(
        integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    integral-type fetch_and(
        integral-type,memory_order = memory_order_seq_cst) noexcept;
    integral-type fetch_or(
        integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    integral-type fetch_or(
        integral-type,memory_order = memory_order_seq_cst) noexcept;
    integral-type fetch_xor(
        integral-type,memory_order = memory_order_seq_cst)
        volatile noexcept;
    integral-type fetch_xor(
        integral-type,memory_order = memory_order_seq_cst) noexcept;
    integral-type operator++() volatile noexcept;
    integral-type operator++() noexcept;
    integral-type operator++(int) volatile noexcept;
    integral-type operator++(int) noexcept;
    integral-type operator--() volatile noexcept;
    integral-type operator--() noexcept;
    integral-type operator--(int) volatile noexcept;
    integral-type operator--(int) noexcept;
    integral-type operator+=(integral-type) volatile noexcept;
    integral-type operator+=(integral-type) noexcept;
    integral-type operator-=(integral-type) volatile noexcept;
    integral-type operator-=(integral-type) noexcept;
    integral-type operator&=(integral-type) volatile noexcept;
    integral-type operator&=(integral-type) noexcept;
    integral-type operator|=(integral-type) volatile noexcept;
    integral-type operator|=(integral-type) noexcept;
    integral-type operator^=(integral-type) volatile noexcept;
    integral-type operator^=(integral-type) noexcept;
};
bool atomic_is_lock_free(volatile const atomic<integral-type>*) noexcept;
bool atomic_is_lock_free(const atomic<integral-type>*) noexcept;
void atomic_init(volatile atomic<integral-type>*,integral-type) noexcept;
void atomic_init(atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_exchange(
    atomic<integral-type>*,integral-type) noexcept;
```

```
integral-type atomic_exchange_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_exchange_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
void atomic_store(volatile atomic<integral-type>*,integral-type) noexcept;
void atomic_store(atomic<integral-type>*,integral-type) noexcept;
void atomic_store_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
void atomic_store_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_load(volatile const atomic<integral-type>*) noexcept;
integral-type atomic_load(const atomic<integral-type>*) noexcept;
integral-type atomic_load_explicit(
    volatile const atomic<integral-type>*,memory_order) noexcept;
integral-type atomic_load_explicit(
    const atomic<integral-type>*,memory_order) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type new_value) noexcept;
bool atomic_compare_exchange_strong(
    atomic<integral-type>*,
    integral-type * old_value,integral-type new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<integral-type>*,
    integral-type * old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type new_value) noexcept;
bool atomic_compare_exchange_weak(
    atomic<integral-type>*,
    integral-type * old_value,integral-type new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
    atomic<integral-type>*,
    integral-type * old_value,integral-type new_value,
    memory_order success_order,memory_order failure_order) noexcept;

integral-type atomic_fetch_add(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_add(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_add_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_add_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_sub(
    volatile atomic<integral-type>*,integral-type) noexcept;
```

```
integral-type atomic_fetch_sub(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_sub_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_sub_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_and(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_and(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_and_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_and_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_or(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_or(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_or_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_or_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_xor(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_xor(
    atomic<integral-type>*,integral-type) noexcept;
integral-type atomic_fetch_xor_explicit(
    volatile atomic<integral-type>*,integral-type, memory_order) noexcept;
integral-type atomic_fetch_xor_explicit(
    atomic<integral-type>*,integral-type, memory_order) noexcept;
```

在主模板中同时提供的那些操作（参见 D.3.8）拥有同样的语义。

### std::atomic<integral-type>::fetch_add 成员函数

载入一个值，并且将其替换为它的值与提供的 i 的值之和。

**声明**

```
integral-type fetch_add(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_add(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

**结果**

获取*this 现存值并且将旧值+i 存储在*this 中。

**返回**

刚刚在存储之前的*this 值。

**引发**

无

注 这是一项对于包括*this 的内存地址的原子的读-修改-写操作。

## std::atomic_fetch_add 非成员函数

从 atomic<integral-type>实例读取值，并将其替换为该值加上提供的 i 值。

**声明**

```
integral-type atomic_fetch_add(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_add(
    atomic<integral-type>* p, integral-type i) noexcept;
```

**结果**

```
return p->fetch_add(i);
```

## std::atomic_fetch_add_explicit 非成员函数

从 atomic<integral-type>实例读取值，并将其替换为该值加上提供的 i 值。

**声明**

```
integral-type atomic_fetch_add_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_add_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

**结果**

```
return p->fetch_add(i,order);
```

## std::atomic<integral-type>::fetch_sub 成员函数

载入一个值，并且将其替换为它的值减去所提供的 i 的值。

**声明**

```
integral-type fetch_sub(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_sub(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

**结果**

获取*this 现存值并且将旧值-i 存储在*this 中。

返回

刚刚在存储之前的*this值。

引发

无

注 这是一项对于包括*this的内存地址的原子级的读-修改-写操作。

## std::atomic_fetch_sub 非成员函数

从 atomic<integral-type>实例读取值，并将其替换为该值减去提供的i值。

声明

```
integral-type atomic_fetch_sub(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_sub(
    atomic<integral-type>* p, integral-type i) noexcept;
```

结果

```
return p->fetch_sub(i);
```

## std::atomic_fetch_sub_explicit 非成员函数

从atomic<integral-type>实例读取值，并将其替换为该值减去提供的i值。

声明

```
integral-type atomic_fetch_sub_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_sub_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

结果

```
return p->fetch_sub(i,order);
```

## std::atomic<integral-type>::fetch_and 成员函数

载入一个值，并将其替换为它的值与所提供的i值按位与的结果。

声明

```
integral-type fetch_and(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_and(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

**结果**

获取*this 现存的值，并将旧值&i 存储在*this 中。

**返回**

刚刚在存储之前的*this 值。

**引发**

无

注 这是一项对于包括*this 的内存地址的原子级的读-修改-写操作。

### std::atomic_fetch_and 非成员函数

从 atomic<integral-type>实例读取值，并将其替换为它的值与所提供的 i 值按位与的结果。

**声明**

```
integral-type atomic_fetch_and(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_and(
    atomic<integral-type>* p, integral-type i) noexcept;
```

**结果**

```
return p->fetch_and(i);
```

### std::atomic_fetch_and_explicit 非成员函数

从 atomic<integral-type>实例读取值，并将其替换为它的值与所提供的 i 值按位与的结果。

**声明**

```
integral-type atomic_fetch_and_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_and_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

**结果**

```
return p->fetch_and(i,order);
```

### std::atomic<integral-type>::fetch_or 成员函数

载入一个值，并将其替换为它的值与所提供的 i 值按位或的结果。

声明

```
integral-type fetch_or(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_or(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

结果

获取*this现存的值，并将旧值|i存储在*this中。

返回

刚刚在存储之前的*this值。

引发

无

注 这是一项对于包括*this的内存地址的原子级的读-修改-写操作。

## std::atomic_fetch_or非成员函数

从atomic<integral-type>实例读取值，并将其替换为它的值与所提供的i值按位或的结果。

声明

```
integral-type atomic_fetch_or(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_or(
    atomic<integral-type>* p, integral-type i) noexcept;
```

结果

```
return p->fetch_or(i);
```

## std::atomic_fetch_or_explicit非成员函数

从atomic<integral-type>实例读取值，并将其替换为它的值与所提供的i值按位或的结果。

声明

```
integral-type atomic_fetch_or_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_or_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

结果

```
return p->fetch_or(i,order);
```

### std::atomic<integral-type>::fetch_xor 成员函数

载入一个值，并将其替换为它的值与所提供的 i 值按位异或的结果。

**声明**

```
integral-type fetch_xor(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type fetch_xor(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

**结果**

获取*this 现存的值，并将旧值^i 存储在*this 中。

**返回**

刚刚在存储之前的*this 值。

**引发**

无

注 这是一项对于包括*this 的内存地址的原子级的读-修改-写操作。

### std::atomic_fetch_or 非成员函数

从 atomic<integral-type>实例读取值，并将其替换为它的值与所提供的 i 值按位异或的结果。

**声明**

```
integral-type atomic_fetch_xor(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type atomic_fetch_xor(
    atomic<integral-type>* p, integral-type i) noexcept;
```

**结果**

```
return p->fetch_xor(i);
```

### std::atomic_fetch_or_explicit 非成员函数

从 atomic<integral-type>实例读取值，并将其替换为它的值与所提供的 i 值按位异或的结果。

**声明**

```
integral-type atomic_fetch_xor_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type atomic_fetch_xor_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

结果

```
return p->fetch_xor(i,order);
```

### std::atomic<integral-type>::operator++前置自增运算符

递增存储在*this 中的值并返回新值。

声明

```
integral-type operator++() volatile noexcept;
integral-type operator++() noexcept;
```

结果

```
return this->fetch_add(1) + 1;
```

### std::atomic<integral-type>::operator++后置自增运算符

递增存储在*this 中的值并返回旧值。

声明

```
integral-type operator++(int) volatile noexcept;
integral-type operator++(int) noexcept;
```

结果

```
return this->fetch_add(1);
```

### std::atomic<integral-type>::operator--前置自增运算符

递增存储在*this 中的值并返回新值。

声明

```
integral-type operator--() volatile noexcept;
integral-type operator--() noexcept;
```

结果

```
return this->fetch_sub(1) - 1;
```

### std::atomic<integral-type>::operator--后置自增运算符

递增存储在*this 中的值并返回旧值。

声明

```
integral-type operator--(int) volatile noexcept;
integral-type operator--(int) noexcept;
```

结果

```
return this->fetch_sub(1);
```

### std::atomic<integral-type>::operator+=复合赋值运算符

将所给的值加到*this中存储的值上，并返回新值。

**声明**

```
integral-type operator+=(integral-type i) volatile noexcept;
integral-type operator+=(integral-type i) noexcept;
```

**结果**

```
return this->fetch_add(i) + i;
```

### std::atomic<integral-type>::operator-=复合赋值运算符

从*this中存储的值减去所给的值，并返回新值。

**声明**

```
integral-type operator-=(integral-type i) volatile noexcept;
integral-type operator-=(integral-type i) noexcept;
```

**结果**

```
return this->fetch_sub(i,std::memory_order_seq_cst) - i;
```

### std::atomic<integral-type>::operator&=复合赋值运算符

将*this中存储的值替换为所给值和存储在*this中的值按位与的结果，并返回新值。

**声明**

```
integral-type operator&=(integral-type i) volatile noexcept;
integral-type operator&=(integral-type i) noexcept;
```

**结果**

```
return this->fetch_and(i) & i;
```

### std::atomic<integral-type>::operator|=复合赋值运算符

将*this中存储的值替换为所给值和存储在*this中的值按位或的结果，并返回新值。

**声明**

```
integral-type operator|=(integral-type i) volatile noexcept;
integral-type operator|=(integral-type i) noexcept;
```

**结果**

```
return this->fetch_or(i,std::memory_order_seq_cst) | i;
```

std::atomic<integral-type>::operator^=复合赋值运算符

将*this 中存储的值替换为所给值和存储在*this 中的值按位异或的结果，并返回新值。

声明

```
integral-type operator^=(integral-type i) volatile noexcept;
integral-type operator^=(integral-type i) noexcept;
```

结果

```
return this->fetch_xor(i,std::memory_order_seq_cst) ^ i;
```

## D.3.11 std::atomic<T*>偏特化

std::atomic 类模板的 std::atomic<T*>偏特化为每一个指针类型提供了原子数据类型，同时带有一整套的操作。

这些std::atomic<T*>的实例不是CopyConstructible和CopyAssignable的，因为这些操作都不能作为一个单一原子操作来进行。

类定义

```
template<typename T>
struct atomic<T*>
{
    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    bool operator=(T*) volatile;
    bool operator=(T*);

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T*,memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T*,memory_order = memory_order_seq_cst) noexcept;
    T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T* load(memory_order = memory_order_seq_cst) const noexcept;
    T* exchange(T*,memory_order = memory_order_seq_cst) volatile noexcept;
    T* exchange(T*,memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
```

```
    memory_order success_order,memory_order failure_order)
    volatile noexcept;
  bool compare_exchange_strong(
    T* & old_value, T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
  bool compare_exchange_weak(
    T* & old_value, T* new_value,
    memory_order order = memory_order_seq_cst) volatile noexcept;
  bool compare_exchange_weak(
    T* & old_value, T* new_value,
    memory_order order = memory_order_seq_cst) noexcept;
  bool compare_exchange_weak(
    T* & old_value, T* new_value,
    memory_order success_order,memory_order failure_order)
    volatile noexcept;
  bool compare_exchange_weak(
    T* & old_value, T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;

  operator T*() const volatile noexcept;
  operator T*() const noexcept;

  T* fetch_add(
    ptrdiff_t,memory_order = memory_order_seq_cst) volatile noexcept;
  T* fetch_add(
    ptrdiff_t,memory_order = memory_order_seq_cst) noexcept;
  T* fetch_sub(
    ptrdiff_t,memory_order = memory_order_seq_cst) volatile noexcept;
  T* fetch_sub(
    ptrdiff_t,memory_order = memory_order_seq_cst) noexcept;

  T* operator++() volatile noexcept;
  T* operator++() noexcept;
  T* operator++(int) volatile noexcept;
  T* operator++(int) noexcept;
  T* operator--() volatile noexcept;
  T* operator--() noexcept;
  T* operator--(int) volatile noexcept;
  T* operator--(int) noexcept;

  T* operator+=(ptrdiff_t) volatile noexcept;
  T* operator+=(ptrdiff_t) noexcept;
  T* operator-=(ptrdiff_t) volatile noexcept;
  T* operator-=(ptrdiff_t) noexcept;
};

bool atomic_is_lock_free(volatile const atomic<T*>*) noexcept;
bool atomic_is_lock_free(const atomic<T*>*) noexcept;
void atomic_init(volatile atomic<T*>*, T*) noexcept;
void atomic_init(atomic<T*>*, T*) noexcept;
T* atomic_exchange(volatile atomic<T*>*, T*) noexcept;
T* atomic_exchange(atomic<T*>*, T*) noexcept;
T* atomic_exchange_explicit(volatile atomic<T*>*, T*, memory_order)
  noexcept;
T* atomic_exchange_explicit(atomic<T*>*, T*, memory_order) noexcept;
```

```
void atomic_store(volatile atomic<T*>*, T*) noexcept;
void atomic_store(atomic<T*>*, T*) noexcept;
void atomic_store_explicit(volatile atomic<T*>*, T*, memory_order)
    noexcept;
void atomic_store_explicit(atomic<T*>*, T*, memory_order) noexcept;
T* atomic_load(volatile const atomic<T*>*) noexcept;
T* atomic_load(const atomic<T*>*) noexcept;
T* atomic_load_explicit(volatile const atomic<T*>*, memory_order) noexcept;
T* atomic_load_explicit(const atomic<T*>*, memory_order) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<T*>*,T* * old_value,T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<T*>*,T* * old_value,T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_weak(
    atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<T*>*,T* * old_value, T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
    atomic<T*>*,T* * old_value, T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add_explicit(
    volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_add_explicit(
    atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub_explicit(
    volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub_explicit(
    atomic<T*>*, ptrdiff_t, memory_order) noexcept;
```

在主模板中同时提供的那些操作（见 D.3.8）拥有同样的语义。

### std::atomic<T*>::fetch_add 成员函数

载入一个值，并且使用标准指针算术规则将其替换为它的值与所给的 i 的值之和，并返回旧值。

**声明**

```
T* fetch_add(
    ptrdiff_t i,memory_order order = memory_order_seq_cst)
```

```
    volatile noexcept;
T* fetch_add(
    ptrdiff_t i,memory_order order = memory_order_seq_cst) noexcept;
```

结果

获取*this 现存值并且将旧值+i 存储在*this 中。

返回

刚刚在存储之前的*this 值。

引发

无

注 这是一项对于包括*this 的内存地址的原子级的读-修改-写操作。

### std::atomic_fetch_add 非成员函数

从 atomic<T*>实例读取值，并使用标准指针算术规则，将其替换为该值加上提供的 i 值。

声明

```
T* atomic_fetch_add(volatile atomic<T*>* p, ptrdiff_t i) noexcept;
T* atomic_fetch_add(atomic<T*>* p, ptrdiff_t i) noexcept;
```

结果

```
return p->fetch_add(i);
```

### std::atomic_fetch_add_explicit 非成员函数

从 atomic<T*>实例读取值，并使用标准指针算术规则，将其替换为该值加上提供的 i 值。

声明

```
T* atomic_fetch_add_explicit(
    volatile atomic<T*>* p, ptrdiff_t i,memory_order order) noexcept;
T* atomic_fetch_add_explicit(
    atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;
```

结果

```
return p->fetch_add(i,order);
```

### std::atomic<T*>::fetch_sub 成员函数

载入一个值，并且使用标准指针算术规则将其替换为它的值减去所给的 i 的值，并返回旧值。

声明

```
T* fetch_sub(
    ptrdiff_t i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
T* fetch_sub(
    ptrdiff_t i,memory_order order = memory_order_seq_cst) noexcept;
```

结果

获取*this现存值并且将旧值-i存储在*this中。

返回

刚刚在存储之前的*this值。

引发

无

注 这是一项对于包括*this的内存地址的原子级的读-修改-写操作。

## std::atomic_fetch_sub 非成员函数

从atomic<T*>实例读取值，并且使用标准指针算术规则，将其替换为它的值减去所给的i的值。

声明

```
T* atomic_fetch_sub(volatile atomic<T*>* p, ptrdiff_t i) noexcept;
T* atomic_fetch_sub(atomic<T*>* p, ptrdiff_t i) noexcept;
```

结果

```
return p->fetch_sub(i);
```

## std::atomic_fetch_sub_explicit 非成员函数

从atomic<T*>实例读取值，并且使用标准指针算术规则，将其替换为它的值减去所给的i的值。

声明

```
T* atomic_fetch_sub_explicit(
    volatile atomic<T*>* p, ptrdiff_t i,memory_order order) noexcept;
T* atomic_fetch_sub_explicit(
    atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;
```

结果

```
return p->fetch_sub(i,order);
```

## std::atomic<T*>::operator++前置自增运算符

使用标准指针算术规则递增存储在*this中的值并返回新值。

声明

```
T* operator++() volatile noexcept;
T* operator++() noexcept;
```

结果

```
return this->fetch_add(i) + i;
```

### std::atomic<T*>::operator++后置自增运算符

使用标准指针算术规则递增存储在*this中的值并返回旧值。

声明

```
T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
```

结果

```
return this->fetch_add(1);
```

### std::atomic<T*>::operator--前置自增运算符

使用标准指针算术规则递增存储在*this中的值并返回新值。

声明

```
T* operator--() volatile noexcept;
T* operator--() noexcept;
```

结果

```
return this->fetch_add(i) + i;
```

### std::atomic<T*>::operator--后置自增运算符

使用标准指针算术规则递增存储在*this中的值并返回旧值。

声明

```
T* operator--(int) volatile noexcept;
T* operator--(int) noexcept;
```

结果

```
return this->fetch_sub(1);
```

### std::atomic<T*>::operator+=复合赋值运算符

使用标准指针算术规则将所给的值加到*this中存储的值上，并返回新值。

声明

```
T* operator+=(ptrdiff_t i) volatile noexcept;
T* operator+=(ptrdiff_t i) noexcept;
```

结果

```
return this->fetch_add(i) + i;
```

### std::atomic<T*>::operator-=复合赋值运算符

使用标准指针算术规则从`*this`中存储的值减去所给的值，并返回新值。

声明

```
T* operator-=(ptrdiff_t i) volatile noexcept;
T* operator-=(ptrdiff_t i) noexcept;
```

结果

```
return this->fetch_sub(i) - i;
```

## D.4 <future>头文件

<future>头文件提供了一些工具，用来处理来自于可能执行在另一个线程上的操作的异步结果。

头文件内容

```
namespace std
{
    enum class future_status {
        ready, timeout, deferred };
    enum class future_errc
    {
        broken_promise,
        future_already_retrieved,
        promise_already_satisfied,
        no_state
    };
    class future_error;

    const error_category& future_category();

    error_code make_error_code(future_errc e);
    error_condition make_error_condition(future_errc e);

    template<typename ResultType>
    class future;

    template<typename ResultType>
    class shared_future;

    template<typename ResultType>
    class promise;

    template<typename FunctionSignature>
    class packaged_task; // no definition provided
```

```
    template<typename ResultType,typename ... Args>
    class packaged_task<ResultType (Args...)>;

    enum class launch {
        async, deferred
    };

    template<typename FunctionType,typename ... Args>
    future<result_of<FunctionType(Args...)>::type>
    async(FunctionType&& func,Args&& ... args);

    template<typename FunctionType,typename ... Args>
    future<result_of<FunctionType(Args...)>::type>
    async(std::launch policy,FunctionType&& func,Args&& ... args);
}
```

## D.4.1 std::future 类模板

std::future 类模板提供了从另一线程等待异步结果的方法，与 std:: promise、std::packaged_task 类模板和 std::async 函数模板联合使用，可以用来提供此异步结果。在任意时刻，只有一个 std::future 实例引用所有给定的异步结果。

std::future 的实例是 MoveConstructible 和 MoveAssignable 的，但不是 CopyConstructible 或 CopyAssignable 的。

**类定义**

```
template<typename ResultType>
class future
{
public:
    future() noexcept;
    future(future&&) noexcept;
    future& operator=(future&&) noexcept;
    ~future();

    future(future const&) = delete;
    future& operator=(future const&) = delete;

    shared_future<ResultType> share();

    bool valid() const noexcept;

    see description get();

    void wait();

    template<typename Rep,typename Period>
    future_status wait_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    future_status wait_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

### std::future 默认构造函数

构造与异步结果没有关联的 std::future 对象。

**声明**

```
future() noexcept;
```

**结果**

构造一个新的 std::future 实例。

**后置条件**

valid()返回 false。

**引发**

无。

### std::future 移动构造函数

从另一个 std::future 对象中构造 std::future 对象，将与另一 std::future 对象关联的异步结果的所有权转移到新构造的实例中。

**声明**

```
future(future&& other) noexcept;
```

**结果**

从 other 移动构造一个新的 std::future 实例。

**后置条件**

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::future 对象。other 没有关联异步结果。this->valid()的返回值与在调用这一构造函数之前 other.valid()会返回的值相同。other.valid()返回 false。

**引发**

无。

### std::future 移动赋值运算符

将于一个 std::future 对象关联的异步结果的所有权转移到另一个对象中。

**声明**

```
future(future&& other) noexcept;
```

**结果**

在 std::future 实例间转移异步状态的所有权。

**后置条件**

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::future 对象。other 没有关联异步结果。在调用前关联至*this 的异步状态（如果有）的所有权被释放，如果这是最后一个引用则状态被销毁。this->valid()的返回值与在调用这一构造函数之前 other.valid()会返回的值相同。other.valid()返回 false。

引发

无。

### std::future 析构函数

销毁 std::future 对象。

声明

```
~future();
```

结果

销毁*this。如果这是对关联至*this 的异步结果（如果有）的最后一个引用，那么销毁该异步结果。

引发

无。

### std::future::share 成员函数

构造新的 std::shared_future 实例，并将关联至*this 的异步结果的所有权转移至这个新构造的 std::shared_future 实例。

声明

```
shared_future<ResultType> share();
```

结果

如同 shared_future<ResultType>(std::move(*this))。

后置条件

如果有调用 share()，那么在调用它之前关联至*this 的异步结果，现在关联至新构造的 std::shared_future 实例。this->valid()返回 false。

引发

无。

### std::future::valid 成员函数

检查 std::future 实例是否关联至异步结果。

声明

```
bool valid() const noexcept;
```

返回

如果*this 已关联至异步结果，返回 true，否则返回 false。

引发

无。

### std::future::wait 成员函数

如果关联至*this 的状态包含延迟函数，调用此延迟函数。否则，一直等待到关联至 std::future 实例的异步结果就绪。

声明

```
void wait();
```

前置条件

this->valid()应返回 true。

结果

如果关联状态包含延迟函数，调用此延迟函数并存储返回值或将引发的异常存储为异步结果。否则，阻塞直到关联至*this 的异步结果就绪。

引发

无。

### std::future::wait_for 成员函数

一直等到关联至 std::future 实例的异步结果就绪，或者直到指定的时间段逝去。

声明

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

前置条件

this->valid()应返回 true。

结果

如果关联至*this 的异步结果包含延迟函数，它是从对 std:async 的调用发起的且尚未开始执行，则立即返回不进行阻塞。否则一直阻塞到关联至*this 的异步结果就绪，或者由 relative_time 指定的时间段逝去。

返回

如果关联至*this 的异步调用包含延迟函数，它是从对 std:async 的调用发起的且尚未开始执行，返回 std::future_status::deferred，如果关联至*this 的异

步结果就绪，返回 std::future_status::ready，如果由 relative_time 指定的时间段逝去则返回 std::future_status::timeout。

注　线程可能会比指定的时间段阻塞得更久。如果可能，逝去时间应由匀速时钟决定。

引发

无。

### std::future::wait_until 成员函数

一直等到关联至 std::future 实例的异步结果就绪，或者到达一个指定的时间。

声明

```
template<typename Clock,typename Duration>
future_status wait_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

前置条件

this->valid()应返回 true。

结果

如果关联至*this 的异步结果包含延迟函数，它是从对 std:async 的调用发起的且尚未开始执行，则立即返回不进行阻塞。否则一直阻塞到关联至*this 的异步结果就绪，或者 Clock::now()返回一个等于或晚于 absolute_time 的时间。

返回

如果关联至*this 的异步调用包含延迟函数，它是从对 std:async 的调用发起的且尚未开始执行，返回 std::future_status::deferred，如果关联至*this 的异步结果就绪，返回 std::future_status::ready，如果 Clock::now()返回一个等于或晚于 absolute_time 的时间则返回 std::future_status::timeout。

注　不能保证调用线程会被阻塞多久，只有当函数返回 std::future_status::timeout，且 Clock::now()返回一个等于或晚于 absolute_time 的时间的时候，线程才会被解锁。

引发

无。

### std::future::get 成员函数

如果关联着的状态包含一个来自对 std::async 调用的延迟函数，调用该函数并

返回值；否则，一直等待到关联至 std::future 实例的异步结果就绪，接着返回存储的值或引发存储的异常。

声明

```
void future<void>::get();
R& future<R&>::get();
R future<R>::get();
```

前置条件

this->valid()应返回 true。

结果

如果关联至*this 的状态包含延迟函数，调用该延迟函数并且返回结果或者传播任何已引发的异常。

否则，一直阻塞到关联至*this 的异步结果就绪。如果结果是存储的异常，引发该异常。否则，返回存储的值。

返回

如果关联的状态包含延迟函数，返回该函数调用的结果。否则，如果 ResultType 是 void，调用正常返回。如果 ResultType 是某些类型 R 的 R&，返回存储的引用。否则，返回存储的值。

引发

由延迟函数引发的异常，或存储在异步结果中的异常，如果有的话。

后置条件

```
this->valid()==false
```

## D.4.2 std::shared_future 类模板

std::shared_future 类模板提供了从另一线程等待异步结果的方法，与 std::promise、std::packaged_task 类模板和 std::async 函数模板联合使用，可以用来提供此异步结果。多个 std::shared_future 实例可以引用同一个异步结果。

std::shared_future 的实例是 CopyConstructible 或 CopyAssignable 的。你也可以从具有相同 ResultType 的 std::future 中移动构造一个 std::shared_furture。

访问给定的 std::shared_future 实例不是同步的。因此多个线程在没有外部同步的情况下访问同一个 std::shared_future 实例是不安全的。但是访问关联状态是同步的，所以多个线程在没有外部同步的情况下，各自访问共享相同的关联状态的 std::shared_future 独立的实例是安全的。

类定义

```
template<typename ResultType>
class shared_future
{
public:
    shared_future() noexcept;
    shared_future(future<ResultType>&&) noexcept;

    shared_future(shared_future&&) noexcept;
    shared_future(shared_future const&);
    shared_future& operator=(shared_future const&);
    shared_future& operator=(shared_future&&) noexcept;
    ~shared_future();

    bool valid() const noexcept;

    see description get() const;

    void wait() const;
    template<typename Rep,typename Period>
    future_status wait_for(
        std::chrono::duration<Rep,Period> const& relative_time) const;

    template<typename Clock,typename Duration>
    future_status wait_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time)
     const;
};
```

### std::shared_future 默认构造函数

构造与异步结果没有关联的 std::shared_future 对象。

声明

```
shared_future() noexcept;
```

结果

构造一个新的 std::shared future 实例。

后置条件

对于新构造的实例，valid()返回 false。

引发

无。

### std::shared_future 移动构造函数

从另一个 std::shared_future 对象中构造 std::shared_future 对象，将与另一 std::shared_future 对象关联的异步结果的所有权转移到新构造的实例中。

声明

```
shared_future(shared_future&& other) noexcept;
```

结果

从 other 移动构造一个新的 std::shared_future 实例。

后置条件

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::shared——future 对象。other 没有关联异步结果。

引发

无。

### std::shared_future 从 std::future 的移动构造函数

从一个 std::future 对象中构造 std::shared_future 对象，将与 std::future 对象关联的异步结果的所有权转移到新构造的实例中。

声明

```
shared_future(std::future<ResultType>&& other) noexcept;
```

结果

从 other 移动构造一个新的 std::shared_future 实例。

后置条件

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::shared_future 对象。other 没有关联异步结果。

引发

无。

### std::shared_future 拷贝构造函数

从另一个 std::shared_future 对象中构造 std::shared_future 对象，因而源和副本都会指向与源 std::shared_future 对象关联的异步结果，如果有的话。

声明

```
shared_future(shared_future const& other);
```

结果

构造一个新的 std::shared_future 实例。

后置条件

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::shared_future 对象和 other。

引发

无。

### std::shared_future 析构函数

销毁 std::shared_future 对象。

声明

```
~shared_future();
```

结果

销毁*this。如果不再有 std::promise 或 std::packaged_task 实例与关联至*this 的异步结果相关联，并且这是对关联至*this 的异步结果的最后一个 std::shared_future，那么销毁该异步结果。

引发

无。

### std::shared_future::valid 成员函数

检查 std::shared_future 实例是否关联至异步结果。

声明

```
bool valid() const noexcept;
```

返回

如果*this 已关联至异步结果，返回 true，否则返回 false。

引发

无。

### std::shared_future::wait 成员函数

如果关联至*this 的状态包含延迟函数，调用此延迟函数。否则，一直等待到关联至 std::shared_future 实例的异步结果就绪。

声明

```
void wait() const;
```

前置条件

this->valid()应返回 true。

结果

来自在共享相同关联状态的 std::shared_future 实例上多线程的 get()和 wait()调用是序列化的。如果关联状态包含延迟函数，首次调用 get()或 wait()会调用此延迟函数并存储返回值或将引发的异常存储为异步结果。

阻塞直到关联至*this 的异步结果就绪。

引发

无。

## std::shared_future::wait_for 成员函数

一直等待到关联至 std::shared_future 实例的异步结果就绪，或者直到指定的时间段逝去。

声明

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time) const;
```

前置条件

this->valid()应返回 true。

结果

如果关联至*this 的异步结果包含延迟函数，它是从对 std:async 的调用发起的且尚未开始执行，则立即返回不进行阻塞。否则一直阻塞到关联至*this 的异步结果就绪，或者由 relative_time 指定的时间段逝去。

返回

如果关联至*this 的异步调用包含延迟函数，它是从对 std:async 的调用发起的且尚未开始执行，返回 std::future_status::deferred，如果关联至*this 的异步结果就绪，返回 std::future_status::ready，如果由 relative_time 指定的时间段逝去则返回 std::future_status::timeout。

注　线程可能会比指定的时间段阻塞得更久。如果可能，逝去时间应由匀速时钟决定。

引发

无。

## std::shared_future::wait_until 成员函数

一直等待到关联至 std::shared_future 实例的异步结果就绪，或者到达一个指定的时间。

声明

```
template<typename Clock,typename Duration>
bool wait_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time) const;
```

前置条件

this->valid()应返回 true。

**结果**

如果关联至*this的异步结果包含延迟函数，它是从对std:async的调用发起的且尚未开始执行，则立即返回不进行阻塞。否则一直阻塞到关联至*this的异步结果就绪，或者Clock::now()返回一个等于或晚于absolute_time的时间。

**返回**

如果关联至*this的异步调用包含延迟函数，它是从对std:async的调用发起的且尚未开始执行，返回std::future_status::deferred，如果关联至*this的异步结果就绪，返回std::future_status::ready，如果Clock::now()返回一个等于或晚于absolute_time的时间则返回std::future_status::timeout。

注 不能保证调用线程会被阻塞多久，只有当函数返回std::future_status::timeout，且Clock::now()返回一个等于或晚于absolute_time的时间的时候，线程才会被解锁。

**引发**

无。

### std::shared_future::get成员函数

如果关联着的状态包含一个来自对std::async调用的延迟函数，调用该函数并返回值。否则，一直等待到关联至std::shared_future实例的异步结果就绪，接着返回存储的值或引发存储的异常。

**声明**

```
void shared_future<void>::get() const;
R& shared_future<R&>::get() const;
R const& shared_future<R>::get() const;
```

**前置条件**

this->valid()应返回true。

**结果**

来自在共享相同关联状态的std::shared_future实例上多线程的get()和wait()调用是序列化的。如果关联状态包含延迟函数，首次调用get()或wait()会调用此延迟函数并存储返回值或将引发的异常存储为异步结果。

一直阻塞到关联至*this的异步结果就绪。如果结果是存储的异常，引发该异常。否则，返回存储的值。

**返回**

如果ResultType是void，正常返回。如果ResultType是某些类型R的R&，返回存储的引用。否则，返回对存储值的const引用。

引发

存储的异常，如果有的话。

### D.4.3 std::packaged_task 类模板

std::packaged_task 类模板打包了函数或其他可调用对象，以便在函数经过 std::packaged_task 实例调用的时候，结果被存储为异步结果，可以通过 std::future 的实例来取得。

std::packaged_task 的实例是 MoveConstructible 和 MoveAssignable 的，但不是 CopyConstructible 或 CopyAssignable 的。

类定义

```
template<typename FunctionType>
class packaged_task; // undefined

template<typename ResultType,typename... ArgTypes>
class packaged_task<ResultType(ArgTypes...)>
{
public:
    packaged_task() noexcept;
    packaged_task(packaged_task&&) noexcept;
    ~packaged_task();

    packaged_task& operator=(packaged_task&&) noexcept;

    packaged_task(packaged_task const&) = delete;
    packaged_task& operator=(packaged_task const&) = delete;

    void swap(packaged_task&) noexcept;

    template<typename Callable>
    explicit packaged_task(Callable&& func);

    template<typename Callable,typename Allocator>
    packaged_task(std::allocator_arg_t, const Allocator&,Callable&&);

    bool valid() const noexcept;
    std::future<ResultType> get_future();
    void operator()(ArgTypes...);
    void make_ready_at_thread_exit(ArgTypes...);
    void reset();
};
```

#### std::packaged_task 默认构造函数

构造 std::packaged_task 对象。

声明

```
packaged_task() noexcept;
```

**结果**

构造没有关联任务和异步结果的 std::packaged_task 实例。

**引发**

无。

## std::packaged_task 从可调用对象的构造函数

构造有关联任务和异步结果的 std::packaged_task 实例。

**声明**

```
template<typename Callable>
packaged_task(Callable&& func);
```

**前置条件**

表达式 func(args...)应有效，这里 args...中的每个元素 args-i 都必须是相应的 ArgTypes...中 ArgType-i 类型的值。返回值必须能够转换为 ResultType。

**结果**

构造 std::packaged_task 实例，带有未就绪的 ResultType 类型的关联异步结果以及 func 副本的 Callable 类型的关联任务。

**引发**

如果构造函数不能为异步结果分配内存，引发 std::bad_alloc 的异常。Callable 的拷贝或移动构造函数引发的任何异常。

## std::packaged_task 从带有分配器的可调用对象的构造函数

构造有关联任务和异步结果的 std::packaged_task 实例，使用所给的分配器来为关联的异步结果和任务分配内存。

**声明**

```
template<typename Allocator,typename Callable>
packaged_task(
    std::allocator_arg_t, Allocator const& alloc,Callable&& func);
```

**前置条件**

表达式 func(args...)应有效，这里 args...中的每个元素 args-i 都必须是相应的 ArgTypes...中 ArgType-i 类型的值。返回值必须能够转换为 ResultType。

**结果**

构造 std::packaged_task 实例，带有未就绪的 ResultType 类型的关联异步结果以及 func 副本的 Callable 类型的关联任务。异步结果和任务的内存是通过分配器 alloc 或其副本来分配的。

引发

如果构造函数不能为异步结果分配内存，引发 std::bad_alloc 的异常。由 Callable 的拷贝或移动构造函数引发的任何异常。

### std::packaged_task 移动构造函数

从另一个 std::packaged_task 对象中构造 std::packaged_task 对象，将与另一 std::packaged_task 对象关联的异步结果的所有权转移到新构造的实例中。

声明

```
packaged_task(packaged_task&& other) noexcept;
```

结果

从 other 移动构造一个新的 std::packaged_task 实例。

后置条件

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::packaged_task 对象。other 没有关联异步结果。

引发

无。

### std::packaged_task 移动赋值运算符

将于一个 std::packaged_task 对象关联的异步结果的所有权转移到另一个对象中。

声明

```
packaged_task& operator=(packaged_task&& other) noexcept;
```

结果

将关联至 other 的异步结果和任务的所有权转移至*this，并且舍弃所有之前的异步结果，如同 std::packaged_task(other).swap(*this)。

后置条件

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::future 对象。other 没有关联异步结果。

引发

无。

### std::packaged_task::swap 成员函数

交换关联至两个 std::packaged_task 对象的异步结果的所有权。

声明

```
void swap(packaged_task& other) noexcept;
```

结果

交换关联至 other 和*this 的异步结果的所有权。

后置条件

在调用 swap 之前关联至 other 的异步结果和任务（如果有）现在关联至*this。在调用 swap 之前关联至*this 的异步结果和任务（如果有）现在关联至 other。

引发

无。

### std::packaged_task 析构函数

销毁 std::packaged_task 对象。

声明

```
~packaged_task();
```

结果

销毁*this。如果*this 拥有关联的异步结果，且该结果没有存储任务或异常，那么此结果变成就绪，带有 std::future_errc::broken_promise 错误码的 std::future_error 异常。

引发

无。

### std::packaged_task::get_future 成员函数

为关联至*this 的异步结果获取 std::future 实例。

声明

```
std::future<ResultType> get_future();
```

前置条件

*this 拥有关联的异步结果。

返回

针对关联至*this 的异步结果的 std::future 实例。

引发

如果 std::future 已经在之前通过调用 get_future()获取过了；引发带有 std::future_errc::future_already_retrieved 错误码的 std::future_error 类型的异常。

### std::packaged_task::reset 成员函数

为同一个任务关联 std::packaged_task 至新的异步结果。

**声明**

```
void reset();
```

**前置条件**

*this 拥有关联的异步任务。

**结果**

如同*this=packaged_task(std::move(f))，这里 f 是已存储的关联至*this 的任务。

**引发**

如果不能为新的异步结果分配内存，引发 std::bad_alloc 的异常。

### std::packaged_task::valid 成员函数

检查*this 是否拥有相关联的已步结果。

**声明**

```
bool valid() const noexcept;
```

**返回**

如果*this 已关联至异步结果，返回 true，否则返回 false。

**引发**

无。

### std::packaged_task::operator()函数调用运算符

调用关联至 std::packaged_task 实例的任务，并且将返回值或异常存储在相关联的异步结果中。

**声明**

```
void operator()(ArgTypes... args);
```

**前置条件**

*this 拥有关联的任务。

**结果**

像 INVOKE(func,args...)那样调用关联的任务 func。如果调用正常地返回，将返回值存储在关联至*this 的异步结果中。如果调用带有异常地返回，将异常存储在关联至*this 的异步结果中。

**后置条件**

关联至*this 的异步结果**就绪**，带有存储的值或异常。所有等待异步结果的被阻塞线程全部解除阻塞。

**引发**

如果异步结果已经拥有了存储的值或异常，引发带有 std::future_errc::promise_already_satisfied 错误码的 std::future_error 类型的异常。

**同步**

成功的对函数调用运算符进行调用，与对 std::future<ResultType>::get() 或 std::shared_future<ResultType>::get() 的调用同步，它们获取已存储的值或异常。

### std::packaged_task::make_ready_at_thread_exit 成员函数

调用关联至 std::packaged_task 实例的任务，并且将返回值或异常存储在相关联的异步结果中，直到线程结束前都不将关联的异步结果变为**就绪**。

**声明**

void make_ready_at_thread_exit(ArgTypes... args);

**前置条件**

*this 拥有关联的任务。

**结果**

像 INVOKE(func,args...)那样调用关联的任务 func。如果调用正常地返回，将返回值存储在关联至*this 的异步结果中。如果调用带有异常地返回，将异常存储在关联至*this 的异步结果中。调度关联的异步状态在当前线程退出的时候变为**就绪**。

**后置条件**

关联至*this 的异步结果拥有存储的值或异常，但直到当前线程推出之前都不是就绪的。所有等待异步结果的被阻塞线程在当前线程退出时全部解除阻塞。

**引发**

如果异步结果已经拥有了存储的值或异常，引发带有 std::future_errc::promise_already_satisfied 错误码的 std::future_error 类型的异常。如果*this 没有相关联的同步状态，引发带有 std::future_errc::no_state 的 std::future_error 类型的异常。

**同步**

成功的对函数调用运算符进行调用，与对 std::future<ResultType>::get() 或 std::shared_future<ResultType>::get() 的调用同步，它们获取已存储的值或异常。

### D.4.4 std::promise 类模板

std::promise 类模板提供了从设置异步结果的方法，可以通过 std::future 实例从另一线程获取它。

ResultType 模板参数是可以被存储在异步结果的值的类型。

关联至特定的 std::promise 实例的异步结果的 std::future 可以通过调用 get_future()成员函数来获得。异步结果既可以用 set_value()成员函数设置为 ResultType 类型的值，也可以使用 set_exception()成员函数设置为一个异常。

std::future 的实例是 MoveConstructible 和 MoveAssignable 的，但不是 CopyConstructible 或 CopyAssignable 的。

**类定义**

```
template<typename ResultType>
class promise
{
public:
    promise();
    promise(promise&&) noexcept;
    ~promise();
    promise& operator=(promise&&) noexcept;

    template<typename Allocator>
    promise(std::allocator_arg_t, Allocator const&);

    promise(promise const&) = delete;
    promise& operator=(promise const&) = delete;
    void swap(promise& ) noexcept;
    std::future<ResultType> get_future();

    void set_value(see description);
    void set_exception(std::exception_ptr p);
};
```

#### std::promise 默认构造函数

构造 std::promise 对象。

**声明**

```
promise();
```

**结果**

构造 std::promise 实例，与一个没有就绪的 ResultType 类型的异步结果相关联。

**引发**

如果构造函数不能为异步结果分配内存，引发 std::bad_alloc 异常。

### std::promise 分配器构造函数

构造 std::promise 对象，使用所给的分配器为关联的异步结果分配内存。

**声明**

```
template<typename Allocator>
promise(std::allocator_arg_t, Allocator const& alloc);
```

**结果**

构造 std::promise 实例，与一个没有就绪的 ResultType 类型的异步结果相关联。通过分配器 alloc 为异步结果分配内存。

**引发**

构造器在试图为异步结果分配内存是引发的所有异常。

### std::promise 移动构造函数

从另一个 std::promise 对象中构造 std::promise 对象，将与另一 std::promise 对象关联的异步结果的所有权转移到新构造的实例中。

**声明**

```
promise(promise&& other) noexcept;
```

**结果**

构造一个新的 std::promise 实例。

**后置条件**

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::promise 对象。other 没有关联异步结果。

**引发**

无。

### std::promise 移动赋值运算符

将于一个 std::promise 对象关联的异步结果的所有权转移到另一个对象中。

**声明**

```
promise& operator=(promise&& other) noexcept;
```

**结果**

将关联至 other 的异步结果的所有权转移给*this。如果*this 已经有了相关联的异步结果，该异步结果变为就绪，带有 std::future_errc::broken_promise 错误码的 std::future_error 类型的异常。

后置条件

在调用此构造函数之前与 other 关联的异步结果，现在被关联至新构造的 std::future 对象。other 没有关联异步结果。

返回

```
*this
```

引发

无。

### std::promise::swap 成员函数

交换关联至两个 std::promise 对象的异步结果的所有权。

声明

```
void swap(promise& other);
```

结果

交换关联至 other 和*this 的异步结果的所有权。

后置条件

在调用 swap 之前关联至 other 的异步结果和任务（如果有）现在关联至*this。在调用 swap 之前关联至*this 的异步结果和任务（如果有）现在关联至 other。

引发

无。

### std::promise 析构函数

销毁 std::promise 对象。

声明

```
~promise();
```

结果

销毁*this。如果*this 拥有关联的异步结果，且该结果没有存储任务或异常，那么此结果变成就绪，带有 std::future_errc::broken_promise 错误码的 std::future_error 异常。

引发

无。

### std::promise::get_future 成员函数

为关联至*this 的异步结果获取 std::future 实例。

声明

```
std::future<ResultType> get_future();
```

前置条件

*this 拥有关联的异步结果。

返回

针对关联至*this 的异步结果的 std::future 实例。

引发

如果 std::future 已经在之前通过调用 get_future()获取过了，引发带有 std::future_errc::future_already_retrieved 错误码的 std::future_error 类型的异常。

## std::promise::set_value 成员函数

将一个值存储在与*this 相关联的异步结果中。

声明

```
void promise<void>::set_value();
void promise<R&>::set_value(R& r);
void promise<R>::set_value(R const& r);
void promise<R>::set_value(R&& r);
```

前置条件

*this 拥有关联的异步任务。

结果

如果 ResultType 不是 void，就将 r 存储在与*this 关联的异步结果中。

后置条件

关联至*this 的异步结果就绪，带有存储的值。所有等待异步结果的被阻塞线程全部解除阻塞。

引发

如果异步结果已经拥有了存储的值或异常，引发带有 std::future_errc::promise_already_satisfied 错误码的 std::future_error 类型的异常。由 r 的拷贝构造函数或移动构造函数引发的所有异常。

同步

多个并发的 set_value()、set_value_at_thread_exit()、set_exception()和 set_exception_at_thread_exit()调用都是序列化的。成功的对 set_value()进行调用，发生于对 std::future<ResultType>::get() 或 std::shared_future<ResultType>::get()之前，它们获取已存储的值。

### std::promise::set_value_at_thread_exit 成员函数

将值存储在与*this 相关联的异步结果中，直到线程结束前都不将关联的异步结果变为就绪。

**声明**

```
void promise<void>::set_value_at_thread_exit();
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<R>::set_value_at_thread_exit(R const& r);
void promise<R>::set_value_at_thread_exit(R&& r);
```

**前置条件**

*this 拥有关联的异步结果。

**结果**

如果 ResultType 不是 void，就将 r 存储在关联至*this 的异步结果中。将异步结果标记为拥有存储的值。调度关联的异步结果在当前线程退出的时候变为就绪。

**后置条件**

关联至*this 的异步结果拥有存储的值，但直到当前线程推出之前都不是就绪的。所有等待异步结果的被阻塞线程在当前线程退出时全部解除阻塞。

**引发**

如果异步结果已经拥有了存储的值或异常，引发带有 std::future_errc::promise_already_satisfied 错误码的 std::future_error 类型的异常。由 r 的拷贝构造函数或移动构造函数引发的所有异常。

**同步**

多个并发的 set_value()、set_value_at_thread_exit()、set_exception() 和 set_exception_at_thread_exit()调用都是序列化的。成功的对 set_value_at_thread_exit()进行调用，发生于对 std::future<ResultType>::get()或 std::shared_future<ResultType>::get()之前，它们获取已存储的值。

### std::promise::set_exception 成员函数

将一个异常存储在与*this 相关联的异步结果中。

**声明**

```
void set_exception(std::exception_ptr e);
```

**前置条件**

*this 拥有关联的异步任务。(bool)e 为 true。

**结果**

将 e 存储在与*this 关联的异步结果中。

后置条件

关联至*this 的异步结果就绪，带有存储的异常。所有等待异步结果的被阻塞线程全部解除阻塞。

引发

如果异步结果已经拥有了存储的值或异常，引发带有 std::future_errc::promise_already_satisfied 错误码的 std::future_error 类型的异常。

同步

多个并发的 set_value()、set_value_at_thread_exit()、set_exception() 和 set_exception_at_thread_exit()调用都是序列化的。成功的对 set_value() 进行调用，发生于对 std::future<ResultType>::get()或 std::shared_future<ResultType>::get()之前，它们获取已存储的异常。

### std::promise::set_exception_at_thread_exit 成员函数

将异常存储在与*this 相关联的异步结果中，直到线程结束前都不将关联的异步结果变为就绪。

声明

```
void set_exception_at_thread_exit(std::exception_ptr e);
```

前置条件

*this 拥有关联的异步结果。(bool)e 为 true。

结果

将 e 存储在关联至*this 的异步结果中。调度关联的异步结果在当前线程退出的时候变为就绪。

后置条件

关联至*this 的异步结果拥有存储的异常，但直到当前线程退出之前都不是就绪的。所有等待异步结果的被阻塞线程在当前线程退出时全部解除阻塞。

引发

如果异步结果已经拥有了存储的值或异常，引发带有 std::future_errc::promise_already_satisfied 错误码的 std::future_error 类型的异常。

同步

多个并发的 set_value()、set_value_at_thread_exit()、set_exception() 和 set_exception_at_thread_exit()调用都是序列化的。成功的对 set_exception_at_thread_exit()进行调用，发生于对 std::future<ResultType>::get()或 std::shared_future<ResultType>::get()之前，它们获取已存储的异常。

## D.4.5　std::async 函数模板

std::async 是一种利用现成的硬件并发来运行自包含异步人物的简单途径。对 std::async 的调用返回一个包含任务结果的 std::future。取决于启动策略，该任务可以异步地运行在它自己的线程上，也可以同步地运行于任何在此 future 调用 wait() 或 get() 成员函数的线程上。

**声明**

```
enum class launch
{
async,deferred
};

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(Callable&& func,Args&& ... args);

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(launch policy,Callable&& func,Args&& ... args);
```

**前置条件**

对于所给的 func 和 args 值，表达式 INVOKE(func,args) 是有效地。Callable 和 Args 的所有成员都是 MoveConstructible 的。

**结果**

在内部存储中构造 func 和 args... 的副本（分别记为 fff 和 xyz...）。

如果 policy 是 std::launch::async，在其自己的线程上运行 INVOKE(fff,xyz...)。所返回的 std::future 会在此线程完成的时候变为就绪，并且会持有返回值或有函数调用所引发的异常。最后一个与 std::future 返回的异步状态同步的 future 对象的析构函数会一直被阻塞到 future 就绪。

如果 policy 是 std::launch::deferred，fff 和 xyz... 会作为延迟函数调用被存储在所返回的 std::future 中。在共享相同关联状态的 future 上首次对 wait() 或 get() 成员函数的调用，会同步地在调用 wait() 或 get() 的线程上执行 INVOKE(fff,xyz...)。

通过执行 INVOKE(fff,xyz...) 所返回的值或引发的异常，会从在该 std::future 上对 get() 的调用中返回。

如果 policy 是 std::launch::async | std::launch::deferred 或 policy 参数被省略，该行为如同 std::launch::async 或 std::launch::deferred 之一被指定。此实现会基于 call-by-call 理论选择具体行为，以便利用可用的硬件并发且没有超量的过度订阅。

在所有情况下，对 std::async 的调用立刻返回。

同步

函数调用的完成，发生于成功从在引用相同关联状态的 std::future 或 std::shared_future 实例上对 wait()、get()、wait_for()或 wait_until()的调用中返回之前，如 std::future 对象从 std::async 的调用中返回。在 std::launch::async 的 policy 情况下，函数调用所在的线程的完成同样发生于成功从这些调用返回之前。

引发

如果无法分配所需的内部存储，引发 std::bad_alloc 异常，否则当无法达成结果或在构造 fff 和 xyz...时引发了任何的异常，就引发 std::future_error 异常。

## D.5 <mutex>头文件

<mutex>头文件提供了担保互斥的功能：互斥元类型、锁类型和函数，以及确保一项操作恰好被执行一次的机制。

头文件内容

```
namespace std
{
    class mutex;
    class recursive_mutex;
    class timed_mutex;
    class recursive_timed_mutex;

    struct adopt_lock_t;
    struct defer_lock_t;
    struct try_to_lock_t;

    constexpr adopt_lock_t adopt_lock{};
    constexpr defer_lock_t defer_lock{};
    constexpr try_to_lock_t try_to_lock{};

    template<typename LockableType>
    class lock_guard;

    template<typename LockableType>
    class unique_lock;

    template<typename LockableType1,typename... LockableType2>
    void lock(LockableType1& m1,LockableType2& m2...);

    template<typename LockableType1,typename... LockableType2>
    int try_lock(LockableType1& m1,LockableType2& m2...);

    struct once_flag;

    template<typename Callable,typename... Args>
    void call_once(once_flag& flag,Callable func,Args args...);
}
```

## D.5.1 std::mutex 类

std::mutex 类为线程提供了基本的互斥与同步机制，可用来保护共享数据。在访问互斥元所保护的数据之前，该互斥元必须通过调用 lock()或 try_lock()来锁定。在同一时刻仅有一个线程可以持有这个锁，如果另一个线程也试图锁定此互斥元，就会失败或被适当地阻塞。一旦线程完成了访问共享数据，它必须接着调用 unlock()来释放锁，并允许其他线程获得它。

std::mutex 满足 Lockable 的需求。

**类定义**

```
class mutex
{
public:
    mutex(mutex const&)=delete;
    mutex& operator=(mutex const&)=delete;

    constexpr mutex() noexcept;
    ~mutex();

    void lock();
    void unlock();
    bool try_lock();
};
```

### std::mutex 默认构造函数

构造 std::mutex 对象。

**声明**

```
constexpr mutex() noexcept;
```

**结果**

构造 std::mutex 实例。

**后置条件**

新构造的 std::mutex 对象初始是未锁定的。

**抛出**

无。

### std::mutex 析构函数

销毁 std::mutex 对象。

**声明**

```
~mutex();
```

前置条件

*this 不得被锁定。

结果

销毁*this。

抛出

无。

### std::mutex::lock 成员函数

为当前线程获取在 std::mutex 对象上的锁。

声明

```
void lock();
```

前置条件

调用线程不得持有*this 上的锁。

结果

阻塞当前线程，直到能够获得*this 上的锁。

后置条件

*this 被调用线程锁定。

抛出

如果有错误发生，抛出 std::system_error 类型的异常。

### std::mutex::try_lock 成员函数

尝试为当前线程获取 std::mutex 对象上的锁。

声明

```
bool try_lock();
```

前置条件

调用线程不得持有*this 上的锁。

结果

尝试为调用线程在非阻塞的情况下获取*this 上的锁。

返回

如果为调用线程获取到锁，返回 true，否则 false。

后置条件

如果函数返回 true，则*this 被调用线程锁定。

抛出

无。

注意：即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。

#### std::mutex::unlock 成员函数

释放当前线程持有的 std::mutex 对象上的锁。

**声明**

```
void unlock();
```

**前置条件**

调用线程必须持有*this 上的锁。

**结果**

释放当前线程持有的*this 上的锁。如果有被阻塞的线程正等待获取*this 上的锁，则对其解除阻塞。

**抛出**

无。

### D.5.2 std::recursive_mutex 类

std::recursive_mutex 类为线程提供了基本的互斥和同步机制，可用来保护共享数据。在访问互斥元所保护的数据之前，该互斥元必须通过调用 lock()或 try_lock()来锁定。在同一时刻仅有一个线程可以持有这个锁，如果另一个线程也试图锁定此 recursive_mutex，就会失败或被适当地阻塞。一旦线程完成了访问共享数据，它必须接着调用 unlock()来释放锁，并允许其他线程获得它。

这里的互斥元是**递归的**（**recursive**），因此持有在特定 std::recursive_mutex 上锁的线程可以进一步调用 lock()或 try_lock()来增加锁定计数值。该互斥元不能被另外的线程锁定，直到获得锁的线程为每个对 lock()和 try_lock()的成功调用都调用过一次 unlock()。

std::recursive_mutex 满足 Lockable 的需求。

**类定义**

```
class recursive_mutex
{
public:
    recursive_mutex(recursive_mutex const&)=delete;
    recursive_mutex& operator=(recursive_mutex const&)=delete;
    recursive_mutex() noexcept;
    ~recursive_mutex();
```

```
    void lock();
    void unlock();
    bool try_lock() noexcept;
};
```

### std::recursive_mutex 默认构造函数

构造 std::recursive_mutex 对象。

**声明**

```
recursive_mutex() noexcept;
```

**结果**

构造 std::recursive_mutex 实例。

**后置条件**

新构造的 std::recursive_mutex 对象初始是未锁定的。

**抛出**

如果不能创建新的 std::recursive_mutex 实例，则抛出 std::system_error 类型的异常。

### std::recursive_mutex 析构函数

销毁 std::recursive_mutex 对象。

**声明**

```
~recursive_mutex();
```

**前置条件**

*this 不得被锁定。

**结果**

销毁*this。

**抛出**

无。

### std::recursive_mutex::lock 成员函数

为当前线程获取在 std::recursive_mutex 对象上的锁。

**声明**

```
void lock();
```

**结果**

阻塞当前线程，直到能够获得*this 上的锁。

后置条件

*this 被调用线程锁定。如果调用线程已经持有*this 上的锁，则锁计数值增加一。

抛出

如果有错误发生，抛出 std::system_error 类型的异常。

### std::recursive_mutex::try_lock 成员函数

尝试为当前线程获取 std::recursive_mutex 对象上的锁。

声明

```
bool try_lock() noexcept;
```

结果

尝试为调用线程在非阻塞的情况下获取*this 上的锁。

返回

如果为调用线程获取到锁，返回 true，否则 false。

后置条件

如果函数返回 true，则已经为调用线程获取了新的*this 上的锁。

抛出

无。

注意：如果调用线程已经持有了*this 上的锁，函数返回 true，且调用线程持有的*this 上锁的计数值增加 1。如果当前线程并未持有*this 上的锁，即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。

### std::recursive_mutex::unlock 成员函数

释放当前线程持有的 std::recursive_mutex 对象上的锁。

声明

```
void unlock();
```

前置条件

调用线程必须持有*this 上的锁。

结果

释放当前线程持有的*this 上的锁。如果这是调用线程所持有的最后一个*this 上的锁，那么若有被阻塞的线程正等待获取*this 上的锁，则对其解除阻塞。

后置条件

调用线程持有的*this 上的锁的计数值减一。

**抛出**

无。

### D.5.3 std::timed_mutex 类

std::mutex 提供了基本的互斥与同步机制，在此之上，std::timed_mutex 类为带超时的锁提供了支持。在访问由互斥元保护的数据之前，该互斥元必须通过调用 lock()、try_lock()、try_lock_for()或 try_lock_until()来进行锁定。如果已经有别的进程持有了锁，那么试图获取锁就会有下面几种情况：失败(try_lock())，阻塞直到锁能够被获取（lock()），阻塞直到锁能被获取或者尝试锁定超时（try_lock_for()或 try_lock_until()）。一旦获得了锁（不管是用哪个函数获取到的），在其他线程可以在互斥元上获得该锁之前，都必须调用 unlock()来释放它。

std::timed_mutex 满足 TimedLockable 的需求。

**类定义**

```
class timed_mutex
{
public:
    timed_mutex(timed_mutex const&)=delete;
    timed_mutex& operator=(timed_mutex const&)=delete;

    timed_mutex();
    ~timed_mutex();
    void lock();
    void unlock();
    bool try_lock();

    template<typename Rep,typename Period>
    bool try_lock_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    bool try_lock_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

#### std::timed_mutex 默认构造函数

构造 std::timed_mutex 对象。

**声明**

```
timed_mutex();
```

**结果**

构造 std::timed_mutex 实例。

后置条件

新构造的std::timed_mutex对象初始是未锁定的。

抛出

如果不能创建新的timed_mutex实例，则抛出std::system_error类型的异常。

### std::timed_mutex 析构函数

销毁std::timed_mutex对象。

声明

```
~timed_mutex();
```

前置条件

*this不得被锁定。

结果

销毁*this。

抛出

无。

### std::timed_mutex::lock 成员函数

为当前线程获取在std::timed_mutex对象上的锁。

声明

```
void lock();
```

前置条件

调用线程不得持有*this上的锁。

结果

阻塞当前线程，直到能够获得*this上的锁。

后置条件

*this被调用线程锁定。

抛出

如果有错误发生，抛出std::system_error类型的异常。

### std::timed_mutex::try_lock 成员函数

尝试为当前线程获取std::mutex对象上的锁。

声明

```
bool try_lock();
```

前置条件

调用线程不得持有*this 上的锁。

结果

尝试为调用线程在非阻塞的情况下获取*this 上的锁。

返回

如果为调用线程获取到锁，返回 true，否则 false。

后置条件

如果函数返回 true，则*this 被调用线程锁定。

抛出

无。

注意：即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。

### std::timed_mutex::try_lock_for 成员函数

尝试为当前线程获取 std::timed_mutex 对象上的锁。

声明

```
template<typename Rep,typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

前置条件

调用线程不得持有*this 上的锁。

结果

在 relative_time 指定的时间内，尝试为调用线程获取*this 上的锁。如果 relative_time.count()为零或负数，此调用会立即返回，如同调用了 try_lock()一样。否则，该调用会一直阻塞，直到获取了锁或者经过了 relative_time 所指定的时间段。

返回

如果为调用线程获得了锁，返回 true，否则 false。

后置条件

如果函数返回 true，则*this 被调用线程锁定。

抛出

无。

注意：即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。线程可能会比指定的时间段阻塞更长的时间。如果可能的话，所经过的时间是有可靠时钟来决定的。

### std::timed_mutex::try_lock_until 成员函数

尝试为当前线程获取 std::timed_mutex 对象上的锁。

**声明**

```
template<typename Clock,typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

**前置条件**

调用线程不得持有*this 上的锁。

**结果**

在 absolute_time 指定的时间之前，尝试为调用线程获取*this 上的锁。如果入口处的 absolute_time<=Clock::now()，此调用会立即返回，如同调用了 try_lock()一样。否则，该调用会一直阻塞，直到获取了锁或者 Clock::now()返回等于或晚于 absolute_time 的时间。

**返回**

如果为调用线程获得了锁，返回 true，否则 false。

**后置条件**

如果函数返回 true，则*this 被调用线程锁定。

**抛出**

无。

注意：即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。调用线程将会被阻塞多长时间是没有保证的，除非函数返回 false 然后 Clock::now()返回等于或晚于 absolute_time 的时间，在这时线程才能解除阻塞。

### std::timed_mutex::unlock 成员函数

释放当前线程持有的 std::timed_mutex 对象上的锁。

**声明**

```
void unlock();
```

**前置条件**

调用线程必须持有*this 上的锁。

**结果**

释放当前线程持有的*this 上的锁。如果有被阻塞的线程正等待获取*this 上的锁，则对其解除阻塞。

**后置条件**

*this 不再被调用线程锁定。

**抛出**

无。

### D.5.4 std::recursive_timed_mutex 类

std::recursive_mutex 提供了基本的互斥与同步机制，在此之上，std::recursive_timed_mutex 类为带超时的锁提供了支持。在访问由互斥元保护的数据之前，该互斥元必须通过调用 lock()、try_lock()、try_lock_for() 或 try_lock_until() 来进行锁定。如果已经有别的进程持有了锁，那么试图获取锁就会有下面几种情况：失败（try_lock()），阻塞直到锁能够被获取（lock()），阻塞直到锁能被获取或者尝试锁定超时（try_lock_for() 或 try_lock_until()）。一旦获得了锁（不管是用哪个函数获取到的），在其他线程可以在互斥元上获得该锁之前，都必须调用 unlock() 来释放它。

这里的互斥元是**递归**的，因此持有在特定 std::recursive_timed_mutex 上锁的线程可以通过任意的锁函数来叠加地锁定该实例。在其他线程能够获取该实例的锁之前，所有现存的锁都必须通过调用相应的 unlock() 进行释放。

std::recursive_timed_mutex 满足 TimedLockable 的需求。

**类定义**

```
class recursive_timed_mutex
{
public:
    recursive_timed_mutex(recursive_timed_mutex const&)=delete;
    recursive_timed_mutex& operator=(recursive_timed_mutex const&)=delete;

    recursive_timed_mutex();
    ~recursive_timed_mutex();

    void lock();
    void unlock();
    bool try_lock() noexcept;

    template<typename Rep,typename Period>
    bool try_lock_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    bool try_lock_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

#### std::recursive_timed_mutex 默认构造函数

构造 std::recursive_timed_mutex 对象。

声明

```
recursive_timed_mutex();
```

结果

构造 std::recursive_timed_mutex 实例。

后置条件

新构造的 std::recursive_timed_mutex 对象初始是未锁定的。

抛出

如果不能创建新的 recursive_timed_mutex 实例，则抛出 std::system_error 类型的异常。

### std::recursive_timed_mutex 析构函数

销毁 std::recursive_timed_mutex 对象。

声明

```
~recursive_timed_mutex();
```

前置条件

*this 不得被锁定。

结果

销毁*this。

抛出

无。

### std::recursive_timed_mutex::lock 成员函数

为当前线程获取在 std::recursive_timed_mutex 对象上的锁。

声明

```
void lock();
```

前置条件

调用线程不得持有*this 上的锁。

结果

阻塞当前线程，直到能够获得*this 上的锁。

后置条件

*this 被调用线程锁定。如果调用线程已经持有*this 上的锁，则锁计数值增加一。

抛出

如果有错误发生，抛出 std::system_error 类型的异常。

### std::recursive_timed_mutex::try_lock 成员函数

尝试为当前线程获取 std::mutex 对象上的锁。

**声明**

```
bool try_lock() noexcept;
```

**前置条件**

调用线程不得持有*this 上的锁。

**结果**

尝试为调用线程在非阻塞的情况下获取*this 上的锁。

**返回**

如果为调用线程获取到锁，返回 true，否则 false。

**后置条件**

如果函数返回 true，则*this 被调用线程锁定。

**抛出**

无。

**注意：** 如果调用线程已经持有了*this 上的锁，函数返回 true，且调用线程持有的*this 上锁的计数值增加一。如果当前线程并未持有*this 上的锁，即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。

### std::recursive_timed_mutex::try_lock_for 成员函数

尝试为当前线程获取 std::timed_mutex 对象上的锁。

**声明**

```
template<typename Rep,typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

**前置条件**

调用线程不得持有*this 上的锁。

**结果**

在 relative_time 指定的时间内，尝试为调用线程获取*this 上的锁。如果 relative_time.count()为零或负数，此调用会立即返回，如同调用了 try_lock()一样。否则，该调用会一直阻塞，直到获取了锁或者经过了 relative_time 所指定的时间段。

**返回**

如果为调用线程获得了锁，返回 true，否则 false。

**后置条件**

如果函数返回 true，则*this 被调用线程锁定。

**抛出**

无。

注意：即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。线程可能会比指定的时间段阻塞更长的时间。如果可能的话，所经过的时间是有可靠时钟来决定的。

## std::recursive_timed_mutex::try_lock_until 成员函数

尝试为当前线程获取 std::timed_mutex 对象上的锁。

**声明**

```
template<typename Clock,typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

**前置条件**

调用线程不得持有*this 上的锁。

**结果**

在 absolute_time 指定的时间之前，尝试为调用线程获取*this 上的锁。如果入口处的 absolute_time<=Clock::now()，此调用会立即返回，如同调用了 try_lock()一样。否则，该调用会一直阻塞，直到获取了锁或者 Clock::now()返回等于或晚于 absolute_time 的时间。

**返回**

如果为调用线程获得了锁，返回 true，否则 false。

**后置条件**

如果函数返回 true，则*this 被调用线程锁定。

**抛出**

无。

注意：即使没有其他的线程持有*this 上的锁，函数也可能获取锁失败（并返回 false）。调用线程将会被阻塞多长时间是没有保证的，除非函数返回 false 然后 Clock::now()返回等于或晚于 absolute_time 的时间，在这时线程才能解除阻塞。

## std::recursive_timed_mutex::unlock 成员函数

释放当前线程持有的 std::timed_mutex 对象上的锁。

**声明**

```
void unlock();
```

**前置条件**

调用线程必须持有*this 上的锁。

**结果**

释放当前线程持有的*this 上的锁。如果有被阻塞的线程正等待获取*this 上的锁，则对其解除阻塞。

**后置条件**

*this 不再被调用线程锁定。

**抛出**

无。

## D.5.5 std::lock_guard 类模板

std::lock_guard 类模板提供了基本的锁所有权包装。将要被锁定的互斥元类型由模板参数 Mutex 指定，且必须满足 Lockable 需求。指定的互斥元被锁定于构造函数中，并被解锁于析构函数中。这就提供了一个简单的为一段代码块锁定一个互斥元的方法，并且确保当离开代码块的时候互斥元被解锁，无论是一次性执行到底，还是使用诸如 break 或 return 这样的控制流语句，或是引发异常来达成的情况。

std::lock_guard 的实例不是 MoveConstructible、CopyConsstructible 或 CopyAssignable 的。

**类定义**

```
template <class Mutex>
class lock_guard
{
public:
    typedef Mutex mutex_type;

    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t);
    ~lock_guard();

    lock_guard(lock_guard const& ) = delete;
    lock_guard& operator=(lock_guard const& ) = delete;
};
```

### std::lock_guard 锁定构造函数

构造锁定所给互斥元的 std::lock_guard 实例。

**声明**

```
explicit lock_guard(mutex_type& m);
```

结果

构造引用所给互斥元的std::lock_guard实例。调用m.lock()。

引发

由m.lock()引发的任何异常。

后置条件

*this拥有在m上的锁。

### std::lock_guard 采纳锁定构造函数

构造拥有所给互斥元上锁的std::lock_guard实例。

声明

```
lock_guard(mutex_type& m,std::adopt_lock_t);
```

前置条件

调用线程必须拥有在m上的锁。

结果

构造引用所给互斥元的std::lock_guard实例，并获取调用线程所持有的m上的锁的所有权。

引发

无。

后置条件

*this拥有调用线程持有的在m上的锁。

### std::lock_guard 析构函数

销毁std::lock_guard实例并解锁相应的互斥元。

声明

```
~lock_guard();
```

结果

为*this构造时所提供的互斥元实例m调用m.unlock()。

引发

无。

## D.5.6 std::unique_lock 类模板

std::unique_lock类模板提供了比std::lock_guard更通用的锁所有权包装。将要被锁定的互斥元类型由模板参数Mutex指定，且必须满足BasicLockable需求。一般

来说，指定的互斥元在构造函数中被锁定并在析构函数中被解锁，尽管可以提供额外的构造函数和成员函数来允许其他的可能性。这就提供了一个简单的为一段代码块锁定一个互斥元的方法，并且确保当离开代码块的时候互斥元被解锁，无论是运行到底，还是使用诸如 break 或 return 这样的控制流语句，或是引发异常来达成的。std::condition_variable 的等待函数要求一个 std::unique_lock< std::mutex>的实例，并且 std::unique_lock 的所有实例化都适合与 std:: condition_variable_any 等待函数的 Lockable 参数一起使用。

如果所给的 Mutex 类型满足 Lockable 需求，那么 std::unique_lock<Mutex> 也满足 Lockable 需求。如果在此之外，所给的 Mutex 类型满足 TimedLockable 需求，那么 std::unique_lock<Mutex>也满足 TimedLockable 需求。

std::unique_lock 的实例是 MoveConstructible 和 MoveAssignable 的，但不是 CopyConsstructible 或 CopyAssignable 的。

**类定义**

```
template <class Mutex>
class unique_lock
{
public:
    typedef Mutex mutex_type;

    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);
    unique_lock(mutex_type& m, adopt_lock_t);
    unique_lock(mutex_type& m, defer_lock_t) noexcept;
    unique_lock(mutex_type& m, try_to_lock_t);

    template<typename Clock,typename Duration>
    unique_lock(
        mutex_type& m,
        std::chrono::time_point<Clock,Duration> const& absolute_time);

    template<typename Rep,typename Period>
    unique_lock(
        mutex_type& m,
        std::chrono::duration<Rep,Period> const& relative_time);

    ~unique_lock();

    unique_lock(unique_lock const& ) = delete;
    unique_lock& operator=(unique_lock const& ) = delete;

    unique_lock(unique_lock&& );
    unique_lock& operator=(unique_lock&& );

    void swap(unique_lock& other) noexcept;

    void lock();
    bool try_lock();
    template<typename Rep, typename Period>
    bool try_lock_for(
        std::chrono::duration<Rep,Period> const& relative_time);
```

```
    template<typename Clock, typename Duration>
    bool try_lock_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
    void unlock();

    explicit operator bool() const noexcept;
    bool owns_lock() const noexcept;
    Mutex* mutex() const noexcept;
    Mutex* release() noexcept;
};
```

### std::unique_lock 默认构造函数

构造没有相关联互斥元的 std::unique_lock 实例。

**声明**

```
unique_lock() noexcept;
```

**结果**

构造没有相关联互斥元的 std::unique_lock 实例。

**后置条件**

```
this->mutex()==NULL, this->owns_lock()==false.
```

### std::unique_lock 锁定构造函数

构造锁定所给互斥元的 std::unique_lock 实例。

**声明**

```
explicit unique_lock(mutex_type& m);
```

**结果**

构造引用所给互斥元的 std::unique_lock 实例。调用 m.lock()。

**后置条件**

```
this->owns_lock()==true, this->mutex()==&m.
```

### std::unique_lock 采纳锁定构造函数

构造拥有所给互斥元上锁的 std::unique_lock 实例。

**声明**

```
unique_lock(mutex_type& m,std::adopt_lock_t);
```

**前置条件**

调用线程必须拥有在 m 上的锁。

结果

构造引用所给互斥元的 std::unique_lock 实例，并获取调用线程所持有的 m 上的锁的所有权。

引发

无。

后置条件

```
this->owns_lock()==true, this->mutex()==&m.
```

## std::unique_lock 延迟锁定构造函数

构造不拥有所给互斥元上锁的 std::unique_lock 实例。

声明

```
unique_lock(mutex_type& m,std::defer_lock_t) noexcept;
```

结果

构造引用所给互斥元的 std::unique_lock 实例。

引发

无。

后置条件

```
this->owns_lock()==false, this->mutex()==&m.
```

## std::unique_lock 尝试锁定构造函数

构造与所给互斥元相关联的 std::unique_lock 实例，并尝试获取该互斥元上的锁。

声明

```
unique_lock(mutex_type& m,std::try_to_lock_t);
```

前置条件

用来实例化 std::unique_lock 的 Mutex 类型必须满足 Lockable 需求。

结果

构造引用所给互斥元的 std::unique_lock 实例。调用 m.try_lock()。

引发

无。

后置条件

this->owns_lock()返回调用 m.try_lock()的结果，this->mutex()==&m。

### std::unique_lock 带有超时时间段的尝试锁定构造函数

构造与所给互斥元相关联的 std::unique_lock 实例，并尝试获取该互斥元上的锁。

**声明**

```
template<typename Rep,typename Period>
unique_lock(
    mutex_type& m,
    std::chrono::duration<Rep,Period> const& relative_time);
```

**前置条件**

用来实例化 std::unique_lock 的 Mutex 类型必须满足 TimedLockable 需求。

**结果**

构造引用所给互斥元的 std::unique_lock 实例。调用 m.try_lock_for(relative_time)。

**引发**

无。

**后置条件**

this->owns_lock() 返回调用 m.try_lock_for() 的结果，this->mutex()==&m。

### std::unique_lock 带有超时时间点的尝试锁定构造函数

构造与所给互斥元相关联的 std::unique_lock 实例，并尝试获取该互斥元上的锁。

**声明**

```
template<typename Clock,typename Duration>
unique_lock(
    mutex_type& m,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

**前置条件**

用来实例化 std::unique_lock 的 Mutex 类型必须满足 TimedLockable 需求。

**结果**

构造引用所给互斥元的 std::unique_lock 实例。调用 m.try_lock_until(absolute_time)。

**引发**

无。

**后置条件**

this->owns_lock() 返回调用 m.try_lock_until() 的结果，this->mutex()==&m。

## std::unique_lock 移动构造函数

将锁的所有权从一个 std::unique_lock 对象转移到新创建的 std::unique_lock 对象。

**声明**

```
unique_lock(unique_lock&& other) noexcept;
```

**结果**

构造 std::unique_lock 实例。如果 other 在调用构造函数前拥有互斥元上的锁，该锁现在由新建立的 std::unique_lock 对象所有。

**后置条件**

对于新构造的 std::unique_lock 对象 x，x.mutex() 等于调用该构造函数前的 other.mutex() 值，且 x.owns_lock() 等于调用该构造函数前 other.owns_lock() 的值。other.mutex()==NULL，other.owns_lock()==false。

**引发**

无。

注　std::unique_lock 对象不是 CopyConstructible 的，所有没有拷贝构造函数，只有这个移动构造函数。

## std::unique_lock 移动赋值运算符

将锁的所有权从一个 std::unique_lock 对象转移到另一个 std::unique_lock 对象。

**声明**

```
unique_lock& operator=(unique_lock&& other) noexcept;
```

**结果**

如果 this->owns_lock() 在此调用前返回 true，调用 this->unlock。如果 other 在此赋值之前拥有在互斥元上的锁，该锁现在由*this 所有。

**后置条件**

this->mutex() 等于调用此赋值前的 other.mutex() 值，且 this->owns_lock() 等于调用此赋值前 other.owns_lock() 的值。other.mutex()==NULL，other.owns_lock()==false。

**引发**

无。

注 std::unique_lock 对象不是 CopyConstructible 的，所有没有拷贝赋值运算符，只有这个移动赋值运算符。

### std::unique_lock 析构函数

销毁 std::unique_lock 实例并解锁相应的互斥元，如果它由被销毁的实例所持有。

**声明**

```
~unique_lock();
```

**结果**

如果 this->owns_lock() 返回 true，调用 this->mutex()->unlock()。

**引发**

无。

### std::unique_lock::swap 成员函数

在两个 std::unique_lock 对象之间交换它们相关联的 unique_lock 的所有权。

**声明**

```
void swap(unique_lock& other) noexcept;
```

**结果**

如果 other 在调用之前拥有互斥元上的锁，该锁现在由*this 所有。

如果*this 在调用之前拥有互斥元上的锁，该锁现在由 other 所有。

**后置条件**

this->mutex()与调用前 other.mutex()的值相等。other.mutex()与调用前 this->mutex()的值相等。this->owns_lock()与调用前 other.owns_lock()的值相等。other.owns_lock()与调用前 this->owns_lock()的值相等。

**引发**

无。

### swap 非成员函数

在两个 std::unique_lock 对象之间交换它们相关联的 unique_lock 的所有权。

**声明**

```
void swap(unique_lock& lhs,unique_lock& rhs) noexcept;
```

**结果**

```
lhs.swap(rhs)
```

**引发**

无。

### std::unique_lock::lock 成员函数

获取与*this 相关联的互斥元上的锁。

**声明**

```
void lock();
```

**前置条件**

this->mutex()!=NULL, this->owns_lock()==false。

**结果**

调用 this->mutex()->lock()。

**引发**

任何由 this->mutex()->lock()引发的异常。如果 this->mutex()==NULL，引发带有 std::errc::operation_not_permitted 错误码的 std::system_error 异常。如果在条目上 this->owns_lock()==true，引发带有 std::errc::resource_deadlock_would_occur 错误码的 std::system_error 异常。

**后置条件**

this->owns_lock()==true。

### std::unique_lock::try_lock 成员函数

试图获取与*this 相关联的互斥元上的锁。

**声明**

```
bool try_lock();
```

**前置条件**

用来实例化 std::unique_lock 的 Mutex 类型必须满足 Lockable 需求。this->mutex()!=NULL, this->owns_lock()==false。

**结果**

调用 this->mutex()->try_lock()。

**返回**

如果对 this_mutex()->try_lock()的调用返回 true，则返回 true，否则 false。

引发

任何由 this->mutex()->try_lock()引发的异常。如果 this->mutex()==NULL，引发带有 std::errc::operation_not_permitted 错误码的 std::system_error 异常。如果在条目上 this->owns_lock()==true，引发带有 std::errc::resource_deadlock_would_occur 错误码的 std::system_error 异常。

后置条件

如果函数返回 true，this->owns_lock()==true，否则 this->owns_lock()==false。

### std::unique_lock::unlock 成员函数

释放与*this 相关联的互斥元上的锁。

声明

```
void unlock();
```

前置条件

this->mutex()!=NULL, this->owns_lock()==true。

结果

调用 this->mutex()->unlock()。

引发

任何由 this->mutex()->unlock()引发的异常。如果在条目上 this->owns_lock()==false，引发带有 std::errc::operation_not_permitted 错误码的 std::system_error异常。

后置条件

this->owns_lock()==false。

### std::unique_lock::try_lock_for 成员函数

在指定时间内试图获取与*this 相关联的互斥元上的锁。

声明

```
template<typename Rep, typename Period>
bool try_lock_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

前置条件

用来实例化 std::unique_lock 的 Mutex 类型必须满足 Lockable 需求。this->mutex()!=NULL, this->owns_lock()==false。

**结果**

调用this->mutex()->try_lock_for(relative_time)。

**返回**

如果对this_mutex()->try_lock_for()的调用返回true，则返回true，否则false。

**引发**

任何由this->mutex()->try_lock_for()引发的异常。如果this->mutex()==NULL，引发带有std::errc::operation_not_permitted错误码的std::system_error异常。如果在条目上this->owns_lock()==true，引发带有std::errc::resource_deadlock_would_occur错误码的std::system_error异常。

**后置条件**

如果函数返回true，this->owns_lock()==true，否则this->owns_lock()==false。

### std::unique_lock::try_lock_until成员函数

在指定时间内试图获取与*this相关联的互斥元上的锁。

**声明**

```
template<typename Clock, typename Duration>
bool try_lock_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

**前置条件**

用来实例化std::unique_lock的Mutex类型必须满足Lockable需求。this->mutex()!=NULL，this->owns_lock()==false。

**结果**

调用this->mutex()->try_lock_until(absolute_time)。

**返回**

如果对this_mutex()->try_lock_until()的调用返回true，则返回true，否则false。

**引发**

任何由this->mutex()->try_lock_until()引发的异常。如果this->mutex()==NULL，引发带有std::errc::operation_not_permitted错误码的std::system_error异常。如果在条目上this->owns_lock()==true，引发带有std::errc::resource_deadlock_would_occur错误码的std::system_error异常。

**后置条件**

如果函数返回true，this->owns_lock()==true，否则this->owns_

lock()==false。

### std::unique_lock::operator bool 成员函数

检查*this 是否拥有互斥元上的锁。

**声明**

```
explicit operator bool() const noexcept;
```

**返回**

this->owns_lock()。

**引发**

无。

**注** 这是运算符的 explicit 版本，因此它仅仅在结果被用作布尔量且不会被视为整型值 0 或 1 的上下文中才会被隐式地调用。

### std::unique_lock::owns_lock 成员函数

检查*this 是否拥有互斥元上的锁。

**声明**

```
bool owns_lock() const noexcept;
```

**返回**

如果*this 拥有互斥元上的锁，返回 true，否则 false。

**引发**

无。

### std::unique_lock::mutex 成员函数

返回与*this 相关联的互斥元，如果有的话。

**声明**

```
mutex_type* mutex() const noexcept;
```

**返回**

如果*this 有相关联的互斥元，返回指向它的指针，否则返回 NULL。

**引发**

无。

### std::unique_lock::release 成员函数

返回与*this 相关联的互斥元，如果有的话，并且释放该关联。

声明

```
mutex_type* release() noexcept;
```

结果

断开互斥元与*this 的关联，并不解锁持有的任何锁。

返回

如果*this 有相关联的互斥元，返回指向它的指针，否则返回 NULL。

后置条件

this->mutex()==NULL，this->owns_lock()==false。

引发

无。

注 如果 this->owns_lock()在调用前返回 true，调用者现在对解锁该互斥元负责。

### D.5.7 std::lock 函数模板

std::lock 函数模板提供了同时锁定多于一个互斥元的方法，避免了在不一致的锁顺序下的死锁结果风险。

声明

```
template<typename LockableType1,typename... LockableType2>
void lock(LockableType1& m1,LockableType2& m2...);
```

前置条件

所给的可锁定对象 LockableType1、LockableType2 等类型必须满足 Lockable 需求。

结果

在每个所给的可锁定对象 m1、m2 等上获取锁，通过未指定的顺序来调用那些类型的 lock()、try_lock()和 unlock()来避免死锁。

后置条件

当前线程拥有每个所给顶的可锁定对象上的锁。

引发

调用 lock()、try_lock()和 unlock()时引发的任何异常。

注 如果异常从 std::lock 的调用中传播出来，那么在本函数中所有已经通过调用 lock()或 try_lock()获取锁的对象 m1、m2 等，都必须为其调用 unlock()。

### D.5.8 std::try_lock 函数模板

std::try_lock 函数模板允许你尝试着一次性锁定一系列的可锁定对象，因此它

们可能全都被锁定也可能都没有被锁定。

**声明**

```
template<typename LockableType1,typename... LockableType2>
int try_lock(LockableType1& m1,LockableType2& m2...);
```

**前置条件**

所给的可锁定对象 LockableType1、LockableType2 等类型必须满足 Lockable 需求。

**结果**

尝试在每个所给的可锁定对象 m1、m2 等上获取锁，通过逐个轮流调用 try_lock()。如果一个对 try_lock()的调用返回 false 或引发异常，已经获取的锁通过在对应的可锁定对象上调用 unlock()来释放。

**返回**

如果所有的锁都获取到（每次调用 try_lock()都返回 true），返回-1，否则返回调用 try_lock()返回 false 的对象的基于零的索引。

**后置条件**

如果函数返回-1，当前线程拥有每个所提供的可锁定对象上的锁。否则，该调用所有已获取的锁都已被释放。

**引发**

调用 try_lock()时引发的任何异常。

注 如果异常从 std::try_lock 的调用中传播出来，那么在本函数中所有已经通过调用 try_lock()获取锁的对象 m1、m2 等，都必须为其调用 unlock()。

## D.5.9 std::once_flag 类

std::once_flag 的实例与 std::call_once 一起使用，以确保特定的函数被严格地调用一次，即便有多个线程同时执行调用。

std::once_flag 的实例不是 CopyConstructible、CopyAssignable、MoveConstructible 和 MoveAssignable 的。

**类定义**

```
struct once_flag
{
    constexpr once_flag() noexcept;

    once_flag(once_flag const& ) = delete;
    once_flag& operator=(once_flag const& ) = delete;
};
```

### std::once_flag 默认构造函数

std::once_flag 默认构造函数构造新的 std::once_flag 实例，其状态指示了所关联的函数尚未被调用。

**声明**

```
constexpr once_flag() noexcept;
```

**结果**

构造新的 std::once_flag 实例，其状态指示了所关联的函数尚未被调用。因为这是一个 constexpr 构造函数，带有静态存储时间段的实例作为静态初始化阶段的一部分被构造，这避免了竞争条件和初始化顺序问题。

## D.5.10 std::call_once 函数模板

std::call_once 与 std::once_flag 一起使用，以确保特定的函数被严格地调用一次，即便有多个线程同时执行调用。

**声明**

```
template<typename Callable,typename... Args>
void call_once(std::once_flag& flag,Callable func,Args args...);
```

**前置条件**

对给定的 func 和 args 值，表达式 INVOKE(func,args)有效。Callable 和 Args 的每个成员都是 MoveConstructible 的。

**结果**

在同一个 std::once_flag 对象上的 std::call_once 调用被序列化。如果在同一个 std::once_flag 对象上之前没有过有效的 std::call_once 调用，参数 func(或其副本)如同通过 INVOKE(func,args)那样被调用，并且 std::call_once 的调用有效当且仅当 func 的调用返回而无异常。如果在同一个 std::once_flag 对象上之前有过有效的 std::call_once，对 std::call_once 的调用会返回而不执行 func。

**同步**

在 std::once_flag 对象上有效的 std::call_once 调用的完成，发生于在同一个 std::once_flag 对象上所有接下来的 std::call_once 调用之前。

**引发**

当结果无法达到或从 func 的调用中传播出任何异常时，引发 std::system_error。

## D.6 <ratio>头文件

<ratio>头文件提供了对编译时有理数数学运算的支持。

头文件内容

```
namespace std
{
    template<intmax_t N,intmax_t D=1>
    class ratio;

    // ratio arithmetic
    template <class R1, class R2>
    using ratio_add = see description;

    template <class R1, class R2>
    using ratio_subtract = see description;

    template <class R1, class R2>
    using ratio_multiply = see description;

    template <class R1, class R2>
    using ratio_divide = see description;

    // ratio comparison
    template <class R1, class R2>
    struct ratio_equal;

    template <class R1, class R2>
    struct ratio_not_equal;

    template <class R1, class R2>
    struct ratio_less;

    template <class R1, class R2>
    struct ratio_less_equal;

    template <class R1, class R2>
    struct ratio_greater;

    template <class R1, class R2>
    struct ratio_greater_equal;

    typedef ratio<1, 1000000000000000000> atto;
    typedef ratio<1, 1000000000000000> femto;
    typedef ratio<1, 1000000000000> pico;
    typedef ratio<1, 1000000000> nano;
    typedef ratio<1, 1000000> micro;
    typedef ratio<1, 1000> milli;
    typedef ratio<1, 100> centi;
    typedef ratio<1, 10> deci;
    typedef ratio<10, 1> deca;
    typedef ratio<100, 1> hecto;
    typedef ratio<1000, 1> kilo;
    typedef ratio<1000000, 1> mega;
    typedef ratio<1000000000, 1> giga;
    typedef ratio<1000000000000, 1> tera;
```

```
    typedef ratio<1000000000000000, 1> peta;
    typedef ratio<1000000000000000000, 1> exa;
}
```

## D.6.1 std::ratio 类模板

std::ratio 类模板为编译时算法提供了一套机制，包括诸如二分之一（std::ratio<1,2>）、三分之二（std::ratio<2,3>）或四十三分之十五（std::ratio<15,43>）这样的有理数值。在C++标准库中它被用来在实例化 std::chrono::duration 类模板时指定时间间隔。

**类定义**

```
template <intmax_t N, intmax_t D = 1>
class ratio
{
public:
    typedef ratio<num, den> type;
    static constexpr intmax_t num= see below;
    static constexpr intmax_t den= see below;
};
```

**需求**

D 不能为零。

**描述**

num 和 den 是分数 N/D 约分到最简的分子和分母。den 永远为正数。如果 N 和 D 符号相同，num 是正的；否则 num 是负的。

**示例**

```
ratio<4,6>::num == 2
ratio<4,6>::den == 3
ratio<4,-6>::num == -2
ratio<4,-6>::den == 3
```

## D.6.2 std::ratio_add 模板别名

std::ratio_add 模板别名提供了在编译时将两个 std::ratio 值相加的机制，使用有理数算法。

**定义**

```
template <class R1, class R2>
using ratio_add = std::ratio<see below>;
```

**前置条件**

R1 和 R2 必须是 std::ratio 类模板的实例化。

**结果**

ratio_add<R1,R2>被定义为 std::ratio 实例的一个别名，它表示了由 R1 和

R2 所代表的分数之和，如果它们的和可以没有溢出地计算出来。如果结果计算溢出，程序就是病态的。在没有算法溢出的情况下，std::ratio_add<R1,R2>应当拥有和 std::ratio<R1::num*R2::den+R2::num*R1::den,R1::den*R2::den>相同的 num 和 den 值。

**示例**

```
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::num == 11
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::den == 15

std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::num == 3
std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::den == 2
```

### D.6.3 std::ratio_subtract 模板别名

std::ratio_add 模板别名提供了在编译时将两个 std::ratio 值相减的机制，使用有理数算法。

**定义**

```
template <class R1, class R2>
using ratio_subtract = std::ratio<see below>;
```

**前置条件**

R1 和 R2 必须是 std::ratio 类模板的实例化。

**结果**

ratio_subtract<R1,R2>被定义为 std::ratio 实例的一个别名，它表示了由 R1 和 R2 所代表的分数之差，如果它们的差可以没有溢出地计算出来。如果结果计算溢出，程序就是病态的。在没有算法溢出的情况下，std::ratio_subtract<R1,R2>应当拥有和 std::ratio<R1::num*R2::den-R2::num*R1::den,R1::den*R2::den>相同的 num 和 den 值。

**示例**

```
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::num == 2
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::den == 15

std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::num == -5
std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::den == 6
```

### D.6.4 std::ratio_multiply 模板别名

std::ratio_multiply 模板别名提供了在编译时将两个 std::ratio 值相乘的机制，使用有理数算法。

**定义**

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

**前置条件**

R1 和 R2 必须是 std::ratio 类模板的实例化。

**结果**

ratio_multiply<R1,R2>被定义为 std::ratio 实例的一个别名，它表示了由 R1 和 R2 所代表的分数之积，如果它们的积可以没有溢出地计算出来。如果结果计算溢出，程序就是病态的。在没有算法溢出的情况下，std::ratio_multiply<R1,R2>应当拥有和 std::ratio<R1::num*R2::num,R1::den*R2::den>相同的 num 和 den 值。

**示例**

```
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::num == 2
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::den == 15

std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::num == 5
std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::den == 7
```

## D.6.5 std::ratio_divide 模板别名

std::ratio_divide 模板别名提供了在编译时将两个 std::ratio 值相除的机制，使用有理数算法。

**定义**

```
template <class R1, class R2>
using ratio_divide = std::ratio<see below>;
```

**前置条件**

R1 和 R2 必须是 std::ratio 类模板的实例化。

**结果**

ratio_multiply<R1,R2>被定义为 std::ratio 实例的一个别名，它表示了由 R1 和 R2 所代表的分数相除的结果，如果此结果可以没有溢出地计算出来。如果结果计算溢出，程序就是病态的。在没有算法溢出的情况下，std::ratio_divide<R1,R2>应当拥有和 std::ratio<R1::num*R2::den,R1::den*R2::num>相同的 num 和 den 值。

**示例**

```
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::num == 5
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::den == 6

std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::num == 7
std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::den == 45
```

## D.6.6 std::ratio_equal 类模板

std::ratio_equal 类模板提供了在编译时比较两个 std::ratio 值是否相等

的机制，使用有理数算法。

**类定义**

```
template <class R1, class R2>
class ratio_equal:
    public std::integral_constant<
        bool,(R1::num == R2::num) && (R1::den == R2::den)>
{};
```

**前置条件**

R1 和 R2 必须是 std::ratio 类模板的实例化。

**示例**

```
std::ratio_equal<std::ratio<1,3>, std::ratio<2,6> >::value == true
std::ratio_equal<std::ratio<1,3>, std::ratio<1,6> >::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<2,3> >::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<1,3> >::value == true
```

## D.6.7 std::ratio_not_equal 类模板

std::ratio equal 类模板提供了在编译时比较两个 std::ratio 值是否不相等的机制，使用有理数算法。

**类定义**

```
template <class R1, class R2>
class ratio_not_equal:
    public std::integral_constant<bool,!ratio_equal<R1,R2>::value>
{};
```

**前置条件**

R1 和 R2 必须是 std::ratio 类模板的实例化。

**示例**

```
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,6> >::value == false
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,6> >::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,3> >::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,3> >::value == false
```

## D.6.8 std::ratio_less 类模板

std::ratio_less 模板提供了在编译时比较两个 std::ratio 值，使用有理数算法。

**定义**

```
template <class R1, class R2>
class ratio_less:
    public std::integral_constant<bool,see below>
{};
```

前置条件

R1 和 R2 必须是 std::ratio 类模板的实例化。

结果

std::ratio_less<R1,R2>派生自 std::integral_constant<bool,value>，这里 value 是(R1::num*R2::den)<(R2::num*R1::den)。如果可能的话，其实现应使用避免溢出的方法来计算结果。如果发生了溢出，则程序就是病态的。

示例

```
std::ratio_less<std::ratio<1,3>, std::ratio<2,6> >::value == false
std::ratio_less<std::ratio<1,6>, std::ratio<1,3> >::value == true
std::ratio_less<
    std::ratio<999999999,1000000000>,
    std::ratio<1000000001,1000000000> >::value == true
std::ratio_less<
    std::ratio<1000000001,1000000000>,
    std::ratio<999999999,1000000000> >::value == false
```

## D.6.9 std::ratio_greater 类模板

std::ratio_greater 模板提供了在编译时比较两个 std::ratio 值，使用有理数算法。

定义

```
template <class R1, class R2>
class ratio_greater:
    public std::integral_constant<bool,ratio_less<R2,R1>::value>
{};
```

前置条件

R1 和 R2 必须是 std::ratio 类模板的实例化。

## D.6.10 std::ratio_less_equal 类模板

std::ratio_less_equal 模板提供了在编译时比较两个 std::ratio 值，使用有理数算法。

定义

```
template <class R1, class R2>
class ratio_less_equal:
    public std::integral_constant<bool,!ratio_less<R2,R1>::value>
{};
```

前置条件

R1 和 R2 必须是 std::ratio 类模板的实例化。

### D.6.11 std::ratio_greater_equal 类模板

std::ratio_greater_equal 模板提供了在编译时比较两个 std::ratio 值，使用有理数算法。

**定义**

```
template <class R1, class R2>
class ratio_greater_equal:
    public std::integral_constant<bool,!ratio_less<R1,R2>::value>
{};
```

**前置条件**

R1 和 R2 必须是 std::ratio 类模板的实例化。

## D.7 <thread>头文件

<thread>头文件提供了用来管理和鉴别线程的服务，以及让当前线程挂起的函数。

**头文件内容**

```
namespace std
{
    class thread;

    namespace this_thread
    {
        thread::id get_id() noexcept;

        void yield() noexcept;

        template<typename Rep,typename Period>
        void sleep_for(
            std::chrono::duration<Rep,Period> sleep_duration);

        template<typename Clock,typename Duration>
        void sleep_until(
            std::chrono::time_point<Clock,Duration> wake_time);
    }
}
```

### D.7.1 std::thread 类

std::thread 类用来管理线程的执行。它提供了开始新线程运行和等待线程执行完毕的方法，以及标识线程的方法和其他管理线程执行的函数。

类定义

```
class thread
{
public:
    // Types
    class id;
    typedef implementation-defined native_handle_type; // optional
    // Construction and Destruction
    thread() noexcept;
    ~thread();

    template<typename Callable,typename Args...>
    explicit thread(Callable&& func,Args&&... args);

    // Copying and Moving
    thread(thread const& other) = delete;
    thread(thread&& other) noexcept;

    thread& operator=(thread const& other) = delete;
    thread& operator=(thread&& other) noexcept;

    void swap(thread& other) noexcept;

    void join();
    void detach();
    bool joinable() const noexcept;

    id get_id() const noexcept;

    native_handle_type native_handle();

    static unsigned hardware_concurrency() noexcept;
};
void swap(thread& lhs,thread& rhs);
```

## std::thread::id 类

std::thread::id的实例标识一个特定的线程的执行。

类定义

```
class thread::id
{
public:
    id() noexcept;
};

bool operator==(thread::id x, thread::id y) noexcept;
bool operator!=(thread::id x, thread::id y) noexcept;
bool operator<(thread::id x, thread::id y) noexcept;
bool operator<=(thread::id x, thread::id y) noexcept;
bool operator>(thread::id x, thread::id y) noexcept;
bool operator>=(thread::id x, thread::id y) noexcept;

template<typename charT, typename traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

注意：标识一个特定的线程执行的 std::thread::id 值，应该与默认构造的 std::thread::id 实例的值和所有代表其他线程执行的值都不相同。

对特定的线程，std::thread::id 值不可预测，并且可能在同一个程序的每次执行中都不一样。

std::thread::id 是 CopyConstructible 和 CopyAssignable 的，因此 std::thread::id 的实例可以自由地复制和赋值。

### std::thread::id 默认构造函数

构造一个 std::thread::id 对象并不表示任何线程的执行。

**声明**

```
id() noexcept;
```

**结果**

构造一个 std::thread::id 实例，包括特别的非任何线程（**not any thread**）的值。

**引发**

无。

注意：所有默认构造的 std::thread::id 实例存储相同的值。

### std::thread::id 相等比较运算符

比较两个 std::thread::id 的实例，看它们是否代表了相同的线程的执行。

**声明**

```
bool operator==(std::thread::id lhs,std::thread::id rhs) noexcept;
```

**返回**

如果 lhs 和 rhs 都表示相同的线程的执行或者都有特别的非任何线程值，返回 true。如果 lhs 和 rhs 代表不同的线程的执行，或者其中一个表示线程的执行，另一个具有特别的非任何线程值，返回 false。

**引发**

无。

### std::thread::id 不等比较运算符

比较两个 std::thread::id 的实例，看它们是否代表了不同的线程的执行。

**声明**

```
bool operator!=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

```
!(lhs==rhs)
```

引发

无。

### std::thread::id 小于比较运算符

比较两个 std::thread::id 的实例，看其中一个是否在线程 ID 值总排序中排在另一个之前。

声明

```
bool operator<(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

如果 lhs 的值在线程 ID 值总排序中出现在 rhs 的值之前，返回 true。如果 lhs!=rhs，lhs<rhs 和 rhs<lhs 中必然有一个返回 true，另一个返回 false。如果 lhs==rhs，那么 lhs<rhs 和 rhs<lhs 都返回 false。

引发

无。

注意：默认构造的 std::thread::id 实例所具有的特别的非任何线程值，它小于任何代表了线程的执行的 std::thread::id 实例。如果两个 std::thread::id 实例相等，那么谁都不小于谁。任意一组不同的 std::thread::id 值都可以构成一个总排序，它在程序的执行过程中是一致的。这个排序在同一个程序的每次执行之间可能会变化。

### std::thread::id 小于或等于比较运算符

比较两个 std::thread::id 的实例，看其中一个是否在线程 ID 值总排序中排在另一个之前或与之相等。

声明

```
bool operator<=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

```
!(rhs<lhs)
```

引发

无。

### std::thread::id 大于比较运算符

比较两个 std::thread::id 的实例，看其中一个是否在线程 ID 值总排序中排在

另一个之后。

声明

```
bool operator>(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

```
rhs<lhs
```

引发

无。

### std::thread::id 大于或等于运算符

比较两个 std::thread::id 的实例，看其中一个是否在线程 ID 值总排序中排在另一个之后或与之相等。

声明

```
bool operator>=(std::thread::id lhs,std::thread::id rhs) noexcept;
```

返回

```
!(lhs<rhs)
```

引发

无。

### std::thread::id 流插入运算符

将表示 std::thread::id 值的字符串写到指定的流中。

声明

```
template<typename charT, typename traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

结果

将表示 std::thread::id 值的字符串插入到指定的流中。

返回

out

引发

无。

注意：字符串表示的格式并未指定。相等的 std::thread::id 实例具有相同的表示，不等的实例具有不同的表示。

## std::thread::native_handle_typetypedef

native_handle_type 是一个到可用于特定平台 API 类型的 typedef。

声明

```
typedef implementation-defined native_handle_type;
```

注 该 typedef 是可选的。若存在，则其实现应当提供一个适用于本地特定平台 API 的类型。

## std::thread::native_handle 成员函数

返回 native_handle_type 类型的值，它代表着与*this 关联的执行线程。

声明

```
native_handle_type native_handle();
```

注 该函数是可选的。若存在，则返回值应该适用于本地特定平台 API。

## std::thread 默认构造函数

构造不与执行线程相关联的 std::thread 对象。

声明

```
thread() noexcept;
```

结果

构造 std::thread 实例，它没有关联的执行线程。

后置条件

对新构造的 std::thread 对象 x，x.get_id()==id()。

引发

无。

## std::thread 构造函数

构造与新的执行线程相关联的 std::thread 对象。

声明

```
template<typename Callable,typename Args...>
explicit thread(Callable&& func,Args&&... args);
```

前置条件

func 和 args 的每个元素必须是 MoveConstructible 的。

结果

构造 std::thread 实例并将它关联至新创建的执行线程。复制或移动 func 和 args 的每个元素到内部存储中，并持续在新的执行线程的整个生命周期。在新的执行线程上执行 INVOKE(copy-of-func,copy-of-args)。

后置条件

对新构造的 std::thread 对象 x，x.get_id()!=id()。

引发

如果不能启动新线程，引发 std::system_error 类型的异常。将 func 或 args 复制进内部存储时引发的任何异常。

同步

对构造函数的调用，发生于在新创建的执行线程上执行给定函数之前。

## std::thread 移动构造函数

将执行线程的所有权从一个 std::thread 对象转移到新创建的 std::thread 对象。

声明

```
thread(thread&& other) noexcept;
```

结果

构造 std::thread 实例。如果 other 在调用构造函数之前拥有一个相关联的执行线程，该执行线程现在关联至新创建的 std::thread 对象。否则，新创建的 std::thread 对象没有相关联的执行线程。

后置条件

对新构造的 std::thread 对象 x，x.get_id() 的与调用构造函数前 other.get_id() 的值相等。other.get_id()==id()。

引发

无。

注 std::thread 对象不是 Copyconstructible 的，所以没有拷贝构造函数，只有这个移动构造函数。

## std::thread 析构函数

销毁 std::thread 对象。

声明

```
~thread();
```

**结果**

销毁*this。如果*this 拥有相关联的执行线程（this->joinable()会返回true），调用 std::terminate()来结束程序。

**引发**

无。

### std::thread 移动赋值运算符

将执行线程的所有权从一个 std::thread 对象转移到另一个 std::thread 对象。

**声明**

```
thread& operator=(thread&& other) noexcept;
```

**结果**

如果在调用之前 this->joinable()返回 true，调用 std::terminate()来结束程序。如果 other 在赋值之前拥有一个相关联的执行线程，该执行线程现在关联至*this。否则，*this 没有相关联的执行线程。

**后置条件**

this->get_id()与调用前 other.get_id()的值相等。other.get_id()==id()。

**引发**

无。

**注** std::thread 对象不是 Copyconstructible 的，所以没有拷贝赋值运算符，只有这个移动赋值运算符

### std::thread::swap 成员函数

在两个 std::thread 对象之间交换它们相关的执行线程的所有权。

**声明**

```
void swap(thread& other) noexcept;
```

**结果**

如果 other 在调用之前拥有一个相关联的执行线程，该执行线程现在关联至*this。否则*this 没有相关联的执行线程。如果*this 在调用之前拥有一个相关联的执行线程，该执行线程现在关联至 other。否则 other 没有相关联的执行线程。

**后置条件**

this->get_id()与调用前 other.get_id()的值相等。other.get_id()与调用前 this->get_id()的值相等。

引发

无。

## swap 非成员函数

在两个 std::thread 对象之间交换它们相关的执行线程的所有权。

声明

```
void swap(thread& lhs,thread& rhs) noexcept;
```

结果

```
lhs.swap(rhs)
```

引发

无。

## std::thread::joinable 成员函数

查询*this 是否拥有关联的执行线程。

声明

```
bool joinable() const noexcept;
```

返回

如果*this 拥有相关联的执行线程，返回 true，否则 false。

引发

无。

## std::thread::join 成员函数

等待与*this 相关联的执行线程结束。

声明

```
void join();
```

前置条件

this->joinable()应返回 true。

结果

阻塞当前线程，直到与*this 关联的执行线程结束。

后置条件

this->get_id()==id()。在调用之前与*this 关联的执行线程已结束。

同步

在调用前与*this 相关联的执行线程执行完毕，发生于对 join()的调用返回之前。

引发

如果无法得到结果，或者 this->joinable()返回 false，引发 std::system_error 异常。

### std::thread::detach 成员函数

分离与*this 相关联的执行线程以结束。

声明

```
void detach();
```

前置条件

this->joinable()应返回 true。

结果

分离与*this 相关联的执行线程。

后置条件

this->get_id()==id()，this->joinable==false。

在调用前与*this 相关联的线程被分离，并且不再拥有相关联的 std::thread 对象。

引发

如果无法得到结果，或者在调用时 this->joinable()返回 false，引发 std::system_error 异常。

### std::thread::get_id 成员函数

返回标识着与*this 关联的执行线程的值。

声明

```
thread::id get_id() const noexcept;
```

返回

如果*this 拥有相关联的执行线程，返回标识着该线程的 std::thread::id 实例。否则返回一个默认构造的 std::thread::id。

引发

无。

### std::thread::hardware_concurrency 静态成员函数

返回在当前硬件上能够并发运行的线程数目的提示。

声明

```
unsigned hardware_concurrency() noexcept;
```

返回

在当前硬件上能够并发运行的线程数目。例如，可能是系统中处理器的数量。当此信息不可用或者没有妥善定义，函数返回 0。

引发

无。

## D.7.2 this_thread 命名空间

std::this_thread 命名空间中的函数在调用线程上运行。

### std::this_thread::get_id 非成员函数

返回一个 std::thread::id 类型的值，标识当前的执行线程。

声明

```
thread::id get_id() noexcept;
```

返回

标识当前线程的一个 std::thread::id 的实例。

引发

无。

### std::this_thread::yield 非成员函数

用来告知类库，调用该函数的线程不需要在调用处运行。通常用在密集的循环中，以避免消耗过多的 CPU 时间。

声明

```
void yield() noexcept;
```

结果

为类库提供一个机会，可以调度其他的事情来替代当前线程。

引发

无。

### std::this_thread::sleep_for 非成员函数

将当前线程的执行挂起一段指定的时间。

声明

```
template<typename Rep,typename Period>
void sleep_for(std::chrono::duration<Rep,Period> const& relative_time);
```

**结果**

阻塞当前线程，直到指定的 relative_time 逝去。

注 线程可能会比指定的时间段阻塞更久。如果可能的话，逝去时间应由匀速时钟来决定。

**引发**

无。

## std::this_thread::sleep_until 非成员函数

挂起将当前线程的执行，直到达到指定的时间点。

**声明**

```
template<typename Clock,typename Duration>
void sleep_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

**结果**

阻塞当前线程，直到指定的 Clock 达到了指定的 absolute_time。

注 没有保证调用线程会被阻塞多久，除非 Clock::now()返回的时间等于或者晚于 absolute_time 的时候线程才会解除阻塞。

**引发**

无。

# C++ 并发编程实战

具有多核的多处理器现已成为标配。C++ 语言的 C++11 版本为多线程应用程序提供了强大的支持，你需要掌握其原理、技巧以及新的并发语言特性，才能独领风骚。

本书帮助你循序渐进地学习用 C++11 编写健壮且优雅的多线程应用程序。你将学习线程内存模型、新的线程支持库，以及基础的线程启动和同步功能。与此同时，你还将学到如何解决并发应用程序中的棘手问题。

**本书具有以下特色：**

- 针对 C++11 新标准编写代码；
- 针对多核多处理器编写程序；
- 用于学习的小例子，用于实践的大例子。

本书适合新接触并发编程的 C++ 程序员，以及曾经使用别的语言、API 或平台编写过多线程代码的程序员阅读。

**作者简介：**

**Anthony Williams** 拥有十余年的 C++ 经验，并且是 BSI C++ 专家组的成员。

"有思想、有深度的指南，从专业人士那儿来的第一手资料。"

——Neil Horlock，Credit Suisse

"简化了 C++ 多线程的黑魔法"。

——Rick Wagner，Red Hat

"读这本书让我头痛，但痛定思痛"。

——Joshua Heyer，Ingersoll Rand

"作者展示了如何将并发变为现实。"

——Roger Orr，OR/2 Limited



MANNING

美术编辑：董志桢

分类建议：计算机／程序设计／ Web 设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-38732-5



ISBN 978-7-115-38732-5

定价：89.00 元